

DOCUMENT RESUME

ED 052 491

24

CG 006 501

AUTHOR Roman, Richard Allan
TITLE Manual for GLURP: A Computer Assisted Instruction Language. Information System for Vocational Decisions.
INSTITUTION Harvard Univ., Cambridge, Mass. Graduate School of Education.
SPONS AGENCY Office of Education (DHEW), Washington, D.C.
REPORT NO Proj-R-33
BUREAU NO BR-6-1819
PUB DATE May 70
GRANT OEG-1-6-061819-2240
NOTE 152p.

EDRS PRICE MF-\$0.65 HC-\$6.58
DESCRIPTORS Career Planning, *Communications, Computer Oriented Programs, *Educational Technology, Information Networks, *Information Systems, Language, *Occupational Guidance, *Programing, Vocational Counseling

ABSTRACT

GLURP is a script-writing language constructed to suit the purposes of the Information System for Vocational Decisions (ISVD). Scripts tell the computer what to say and how to act; they allow the computer to interact with whoever is using the system. Teachers, counselors, etc. wishing to structure the learning of their students and clients need to become increasingly skillful in script-writing so that the computer-inquirer interaction is most productive. This manual is designed to prepare people to write in GLURP. It is divided into Beginner, Intermediate and Advanced sections so that gradual, comprehensive learning can take place. (TL)

ED052491

INFORMATION SYSTEM FOR VOCATIONAL DECISIONS

Project Report No. 33

MANUAL FOR GLURP:
A COMPUTER ASSISTED INSTRUCTION LANGUAGE

Richard Allan Roman

This paper was supported in part by Grant No. OEG-1-6-061819-2240
of the United States Office of Education under terms of the
Vocational Education Act of 1963.

Graduate School of Education
Harvard University

May 1970

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRO-
DUCED EXACTLY AS RECEIVED FROM
THE PERSON OR ORGANIZATION ORIG-
INATING IT. POINTS OF VIEW OR OPIN-
IONS STATED DO NOT NECESSARILY
REPRESENT OFFICIAL OFFICE OF EDU-
CATION POSITION OR POLICY.

PREFACE

The script-writing language described in this manual was designed as part of the work of the Information System for Vocational Decisions (ISVD). The work was supported under Grant No. OEG-1-6-0161819-2240 of the United States Office of Education under terms of the Vocational Education Act of 1963.

After investigating existing languages for implementing interactive educational procedures, we decided no one language provided all the features we required. Therefore, we had to construct one to suit our purpose. GLURP combines features of several languages: ELIZA, a pattern recognizing language for conversational interaction (Weizenbaum, 1966); TRAC, a procedure describing language (Mooers, 1966); PLANIT, a language for the lesson designer (SDC, 1966); and finally MINORCA, an earlier in-house language. We believe that GLURP usefully combines features of each of these languages to produce an effective computer assisted instruction language.

No work on GLURP should fail to recognize the contribution of Mrs. Ann Taylor who coded the language and created many of its ideas.

Susan Roman and Robert Aylmer, two experienced GLURP writers have read portions of this manual and have given valuable suggestions. Any remaining errors are the sole responsibility of the author.

This manual has not been tested with naive script writers, so we have no indication of its effectiveness.

The author would appreciate any comments about the manual from anyone who uses it. Of particular interest would be solutions to the problems posed in Appendix A.

Please address comments to

Richard Allan Roman

Learning Research and Development Center
University of Pittsburgh
Pittsburgh, Pennsylvania

TABLE OF CONTENTS

PREFACE	I
TABLE OF CONTENTS	IV
LIST OF APPENDICES	
LIST OF FIGURES	
CHAPTER 1. INTRODUCTION	1
Section A. How To Read the Manual	1
B. The Console: What An Inquirer Sees	2
C. The Script: What An Author Produces	6
CHAPTER 2. NORMAL GLURP FUNCTIONS	11
Section A. GOTO stepno	11
B. KEYBOARD or KEYBOARD:stepno	13
B1 Beginner	13
B2 Intermediate	15
C. CRT Text	17
C1 Beginner	17
C2 Other CRT Statements (Intermediate)	18
C3 Formatting the CRT (Advanced)	21
D. DECOMP	22
D1 Basic DECOMP (Beginner)	22
a. The Exact Match	22
b. Multiple Rules in Same DECOMP	24
c. The Zero (0) in DECOMP	25
d. The Synonym List	27
e. The Dollar Sign (\$) for Suffixes	28
f. Phrases in DECOMP	31
g. External Lists of Phrases	31

D2	The Use of Lists (Intermediate)	33
D3	Tricky Details of DECOMP (Advanced)	36
a.	The Asterisk in External Lists	36
b.	Multiple Sentence Input	37
CHAPTER 3. WEIRD FUNCTIONS IN GLURP		38
Section A. Beginner		38
A1	Introduction	38
A2	Slides	39
A3	Print String	40
A4	Inquirer Characteristics	41
A5	Equals	43
A6	Define String	45
A7	Arithmetic	47
A8	Compare	49
A9	Inquirer Response	50
A10	Miscellaneous Other Useful Information	52
<i>END OF MATERIAL FOR BEGINNERS</i>		
B.	Intermediate	53
B1	Introductory Concepts	53
a.	Nesting Functions	53
b.	Protective Parentheses	54
c.	Names and Values in Arithmetic Functions	58
B2	Forms Storage Manipulation	61
a.	Forms Pointer	61
b.	Segment Gaps	61
c.	Forms Pointer Functions	64

B3	List Manipulation	66
a.	Introduction	.66
b.	New Top	66
c.	New Bottom	.66
d.	Pop Top	67
e.	Delete Item	67
f.	Using the List Functions	.67
B4	List Comparisons	.69
a.	Introduction	.69
b.	Compare for Equalities	.69
c.	Compare for Inequalities	.69
B5	Miscellaneous Forms Manipulation	.71
a.	Concatenate	71
b.	Delete Definition	71
c.	Selective Delete	.71
B6	Printing Functions	.72
a.	Print String	.72
b.	Print Continuous	.72
c.	Print List	.72
B7	Branching Functions	73
a.	Introduction	.73
b.	Go	.73
c.	Reset Step	.74
d.	Execute	75
B8	Answers and DECOMP	.76
a.	Introduction	.76
b.	Load Keyboard	76

B9	Utility Functions	77
B10	Miscellaneous Comments	.80
<i>END OF MATERIAL FOR INTERMEDIATES</i>		
C.	Advanced Weird	.81
C1	Neutral Functions	81
C2	The TRAC Algorithm	.86
C3	Use of Neutral Functions	.87
C4	Binary Strings	.90
C5	Using Binary Functions	.95
C6	Storage -- Changing Data on the Disc	.97
a.	Introduction	.97
b.	Store Record	.98
c.	Other Storage Functions	99
d.	Encoding Data	.100
C7	Fetching Data from the Disc	.103
C8	Special Functions for Debugging	.106
C9	Special Functions Not for Debugging	.108
APPENDICES		.109
A.	Script Topics and Solutions	.109
B.	Flow Chart of CRT Decisions in GLURP	115
C.	Multi-Punches For Sanders Formatting Characters	.119
D.	ISVD Rule Lists	.123
E.	The Weird Functions	.127
F.	The TRAC Algorithm	141
BIBLIOGRAPHY		.149
INDEX		151

FIGURES VII

1. Special Student Commands for ISVD	5
2. A First Script	6
3. CRT Displays from Figure 2	8
4. Correct and Incorrect Step Numbers	11
5. A GOTO Script	12
6. A @HELP Script for Beginners	14
7. A @HELP Script for Intermediates	15
8. A Better @HELP Script for Intermediates	16
9. An Intermediate CRT Script	18
10. The Screen for Figure 9	19
11. Beginner's DECOMP Script	22
12. A Script with Multiple Rules in DECOMP	25
13. A Script Using Dollar Suffixes	28
14. Contextual Analysis with DECOMP	30
15. Branches for Deeper Analysis	30
16. External Lists of Phrases	31
17. Script with DECOMP Lists	34
18. Inquirer Characteristics	42
19. Script Using *stepno	44
20. Use of Author Defined Functions	46
21. Script Using Keyboard Segment	51
22. Script Illustrating Linkage	80
23. A Script Requiring Neutral Strings	87
24. A Segment of EDIT	88

25.	Script Changing with Inquirer Experience	.91
26.	Diagram of File Structure	97
27.	Decode Table - COLFACT - List 16	101
28.	Educational Institutions Data Base - List 1	.104

MANUAL FOR GLURP:

A COMPUTER ASSISTED INSTRUCTION LANGUAGE

INTRODUCTION

Section A How to Read the Manual

Beginner

People learning a new computer language (particularly their first) are usually overwhelmed by the complexity. GLURP can be frightening. Fortunately, one does not have to know everything in this manual to write in GLURP. Beginners can write basic GLURP after reading those sections marked Beginner. Those who want to know more of the capabilities can read the intermediate sections marked Intermediate. Those learning their first computer language should skip sections marked Intermediate on first reading. Finally, those who want to know all the technical details and the specifications for the ISVD implementation of GLURP can read the advanced sections marked Advanced.

Anyone without previous knowledge of GLURP should assume he is a beginner. The intermediate sections presuppose full knowledge of the beginner sections and will confuse a reader without the prerequisite familiarity. Write at least one script using only Beginner's GLURP. Then on a second reading, interested readers could assimilate both the intermediate and advanced sections.

With GLURP, as with any language, you will not learn by passively reading this manual. You must work through the examples, trying to anticipate the discussion. For best results, I would recommend writing a simple script after reading the beginner's section on Regular Functions of GLURP. The script should be revised after completing the section for beginners on "Weird Functions in GLURP". Only by writing material yourself will you understand what GLURP is. To aid your work, Appendix A suggests several topics for scripts.

Section B THE CONSOLE: What an Inquirer Sees

Beginner

To follow this manual, you must know something about the use of GLURP. You will use GLURP to prepare lesson plans which are called 'scripts'. We use the word 'scripts' because the lessons tell the computer what to say and how to act. Once you write a script, it allows the computer to interact with a student or inquirer in a manner approximating your own teaching style. The quality of the approximation is determined by your ability as a scriptwriter. The more of yourself the script contains, the more realistic the interaction will be for the inquirer.

Perhaps you are asking how the computer can teach even with your script inside. First it needs a way of communicating with the inquirer. This can be done through a typewriter controlled by the computer, through slides, or through a TV screen. The TV screen, called a CRT, displays letters and words under computer direction. Words are written very rapidly, far faster than people can read. Essentially the letters appear instantaneously.

Once words are on the screen, the machine signals that the inquirer should read the material and respond. In the ISVD system, three asterisks (***) at the bottom of the CRT screen indicate the computer is ready for a response.

Next the inquirer needs some way to respond to the machine. Several ways are available in today's technology. Devices include handwriting recognizers, light pens with which an inquirer can point to his response, and arrays of buttons like typewriters. On the ISVD system, the inquirer types his answer into the machine. He can read what he types since it is displayed on the CRT under the machine's text. He can correct errors or add things if he wishes. Eventually he will finish and tell the computer to read his answer. In the ISVD, we say he "pushes SEND BLOCK" to say he is finished responding. SEND BLOCK is the name of one of the special buttons added to the typewriter.

Typically the machine would read and process his answers and decide (using your directions) which thing to say next. The text already on the CRT is erased and new material appears. If you think it advantageous, you can instruct the computer to add the new text to the old rather than erase the screen.

Not only do you as author control the interaction, but also any time that the inquirer wants to change what he is doing, he can respond with a command instead of the expected answer. Commands allow an inquirer to ask for clarification if he is confused, ask for specific information at any time and otherwise manipulate the system. As an author, you do not need to be concerned about this; the computer will automatically recognize commands. The list of commands for ISVD is included as Figure 1. These commands suited the ISVD well, but different groups at different installations may want to use other commands.

FIGURE 1

SPECIAL STUDENT COMMANDS FOR ISVD
(Hold down shift key and type P for @)

@HELP	When you don't know how to answer a question
@STOP	When you want to go on to something else
@QUIT	When you want to leave ISVD entirely
@DATA	When you want to know about a specific college, job, trade school, or military job
@DATA-COL	When you want to know about a specific college
@DATA-JOB	When you want to know about a specific job
@DATA-TRADE	When you want to know about a specific trade school
@DATA-MIL	When you want to know about a specific military job
@SUMMARY	When you want a review of what you've been doing

Section C The Script: What an Author Produces

Beginner

You have read about what a script-controlled interaction looks like from the inquirer's viewpoint. What does a script look like to an author? At first glance, a script is just a bunch of gibberish. It looks, in part, like this:

FIGURE 2

A FIRST SCRIPT

```
.  
. .  
. .  
100.00 CRT WHICH OF THESE WILL YOU DO NOW?  
          A. GET INFORMATION ABOUT COLLEGES  
          B. GET INFORMATION ABOUT JOBS  
110.00 KEYBOARD  
120.00 DECOMP:A,B:200.00:300.00  
130.00 CRT+ PLEASE TYPE A OR B.  
140.00 GOTO 110.00  
. .  
. .  
. .
```

The script segment above (Figure 2) contains most of the functions you need to write simple scripts. Let's examine this script part by part. First, notice that the basic unit of a script is the numbered step. In the script segment above, there are five steps. Each step has a number that identifies it. Numbers in this script segment are 100.00, 110.00, 120.00, 130.00, and 140.00. Every step in a single script must have a different number. Step numbers must increase throughout the script. Notice that there are several kinds of steps as indicated by the first word after the number. The step types illustrated in the script segment are CRT, KEYBOARD, DECOMP, CRT+, and GOTO. Each of these will be discussed in detail later. For now, let's follow the general drift of this script segment.

In broad overview, this script section asks a question (step 100.00) and tells the inquirer to respond (step 110.00). It accepts his answer and judges whether it is 'A' or 'B' (step 120.00). If so, it goes

on to the next question (i.e. step 200.00 or 300.00); if not, it asks him to try again (steps 130.00 and 140.00).

The computer has the entire script inside, but it looks at only one step at a time. Each step tells the computer what to do. Normally steps follow each other in order, just as speeches do in the script for a play. However, there are exceptions to this general rule. A step can direct the computer to take the next step out of order. For example, after step 140.00, which says 'GOTO 110.00', the computer will proceed to step 110.00. Generally steps proceed sequentially, but certain instructions cause them to go out of order. With this understanding, let us go on to a step by step analysis of the script segment.

The first step we encounter is number 100.00. CRT, the step type for step 100.00, directs the computer to display the rest of the step onto the CRT for the inquirer to read. The screen is much like a television screen, and is called a cathode ray tube. We get the initials CRT from the words 'cathode ray tube'. After the computer completes step 100.00, the text will be visible to the inquirer on his CRT. He can then read it while the computer proceeds to the next step, 110.00. After step 100.00, the CRT will look like Figure 3. [p. 8]

Step 110.00 is a KEYBOARD type. This directs the computer to allow the inquirer an opportunity to respond to the text displayed on the CRT. The computer tells the inquirer to respond by printing three asterisks (***) near the bottom of the CRT. See Figure 3, Part B. for the screen appearance after step 110.00. The inquirer would not perceive a pause between steps 100.00 and step 110.00. When the inquirer has finished reading the material on the screen, he will type in some response and push SEND BLOCK, the special button that tells the computer that he is done.

Let us assume the inquirer typed in 'COLLEGE'. The screen would then look as it does in Figure 3, part C. When the inquirer pushes SEND BLOCK the computer receives the response and proceeds to the next step, 120.00. Step 120.00 is a DECOMP type step. These direct the computer to look at the inquirer's response and match it against each rule in the step. Rules in a DECOMP step are separated by commas (,); in this case, there are two rules 'A' and 'B'. Notice that there are two step numbers

-8-

Part A After 100.00

WHICH OF THESE WILL YOU DO NOW?
A. GET INFORMATION ABOUT COLLEGES
B. GET INFORMATION ABOUT JOBS

Part B After 110.00

WHICH OF THESE WILL YOU DO NOW?
A. GET INFORMATION ABOUT COLLEGES
B. GET INFORMATION ABOUT JOBS

Part C After first response

WHICH OF THESE WILL YOU DO NOW?
A. GET INFORMATION ABOUT COLLEGES
B. GET INFORMATION ABOUT JOBS

*** COLLEGE

Part D After 130.00

WHICH OF THESE WILL YOU DO NOW?
A. GET INFORMATION ABOUT COLLEGES
B. GET INFORMATION ABOUT JOBS
PLEASE TYPE A OR B.

*** COLLEGE

Part E After second response

WHICH OF THESE WILL YOU DO NOW?
A. GET INFORMATION ABOUT COLLEGES
B. GET INFORMATION ABOUT JOBS
PLEASE TYPE A OR B.

*** B

FIGURE 3

CRT DISPLAYS FROM FIGURE 2

on the right of the DECOMP step, namely 200.00 and 300.00. These correspond to the two rules in the step; 'A' and '200.00' go together as do 'B' and '300.00'. For each rule there is a number to the right. Rules and numbers correspond by being in the same order left to right.

The rules have meaning. They direct the computer to compare the inquirer's response to the contents of the rule; Did the inquirer type in exactly 'A'? Did the inquirer type in exactly 'B'? These rules are matched against the inquirer's response ('COLLEGE' in this case). Had the inquirer's response matched one of the rules, the next step would be the corresponding step number. In this case, if the rule 'A' matched, the next step would be 200.00 and if 'B' matched, the next step would be 300.00. Neither rule matches the response 'COLLEGE', so the computer simply proceeds to the next step in normal sequence, 130.00.

The CRT+ type step now adds new text to that already on the CRT. In this case the step tells the inquirer the appropriate responses, A or B. After displaying the text, the computer proceeds to step 140.00. See Figure 3, part D for the appearance of the screen after step 130.00.

Step 140.00 is a GOTO type, telling the computer to take 110.00 as the next step. At 110.00, a KEYBOARD type step, the inquirer's previous response is erased and he is given the opportunity to respond as in Figure 3, part E. The machine proceeds to step 120.00. Again the DECOMP step analyzes and matches the response against the rules. A match occurs this time with rule 'B', which is the second step number. By proceeding to step 300.00 because of the match, the computer gets outside of our script segment.

Before going on to discuss each of the different kinds of steps in detail, we should think about the implications of scriptwriting for a moment. As you have seen, the DECOMP statement allows you to test what the inquirer has said and change your course of instruction to fit his statement. Errors that an inquirer makes often clearly point to a difficulty in his thinking. By using the DECOMP wisely, you can clarify points of difficulty when they occur. Through DECOMP statements, you can make the instruction quite personal. Notice that not all inquirers need to see all the material written in any script. The DECOMP statement allows you to determine if certain material is appropriate for the particular

inquirer and to avoid boring him if he already has the knowledge. Similarly, you can use DECOMP to determine if an inquirer is missing certain prerequisite information and supply it only if needed. The DECOMP statement is at the heart of the GLURP language. Used wisely and well, it can make computerized instruction lively, personal conversation. Used sloppily or casually, it is just a front for multiple choice instruction or a time waster.

At this point we will discuss each of these step types in more detail. After this discussion, a beginning scriptwriter should be able to write in GLURP. He will be able to write most teaching strategies used in computer-aided instruction. In each of the descriptions, there are beginning, intermediate, and advanced sections. Read only those for which you are ready. There is absolutely no need for a beginner in GLURP to know the intermediate material or for the intermediate to know the advanced material. Remember the right top of the page tells who the material is for.

NORMAL GLURP FUNCTIONS

Section A GOTO stepno

Beginner

The simplest step type is GOTO. The first five characters are 'GOTO '. Notice that the space is essential. After the step type indicator ('GOTO ') comes the step number to which the computer goes next. The symbol 'stepno' here and elsewhere in the manual means a step number in the form ddd.dd where each d is a digit 0, 1, ..., 9. All five of the digits must be written even when they are zeros. Examples of correct and incorrect step numbers follow:

FIGURE 4

CORRECT AND INCORRECT STEP NUMBERS

<u>CORRECT</u>	<u>INCORRECT</u>	
130.00	12031	no decimal point
200.00	212.	missing zeros
115.32	347	missing zeros and decimal point
103.21	3.00	missing leading zeros
002.43	23.215	too many digits after decimal and too few before it
000.01	1132.45	too many digits before the decimal
999.99		

When the computer encounters a GOTO step, it changes the order in which it looks at other steps. Instead of proceeding normally to the next step in sequence, the computer changes course and continues from the step called 'stepno'. As a quick, silly example, consider this piece of script:

FIGURE 5

A GOTO SCRIPT

```
.  
. .  
. .  
100.00 GOTO 150.00  
110.00 GOTO 300.00  
120.00 GOTO 130.00  
120.25 GOTO 215.00  
130.00 GOTO 300.00  
140.00 GOTO 120.00  
150.00 GOTO 120.25  
200.00 GOTO 140.00  
215.00 GOTO 120.00  
280.00 GOTO 100.00  
300.00 CRT WHEW  
. .  
. .  
. .
```

After beginning at 100.00, the computer will eventually print 'WHEW' on the CRT. Be sure you see how it gets there. Only seven of the eleven steps are used. If you are uncertain about that example, the order of the steps as the computer does them is 100.00, 150.00, 120.25, 215.00, 120.00, 130.00, and finally 300.00.

When you use a GOTO type step, you must be certain that the script has a step numbered 'stepno'. You can not GOTO a step that does not exist in the finished script. As you write, you may write GOTO instructions to steps that do not exist yet, but you must be certain that you eventually include a step with that number.

One technique for keeping track of stepno's used in your GOTO steps (branches in more formal language) has proved its utility. As soon as you write a branch (GOTO) to a step not yet written, list the number of the step and a brief statement about what it will do. As you proceed, refer to the list and scratch off step numbers as you write them. When the script is finished, your list should be empty.

Section B1 KEYBOARD

KEYBOARD: stepno

The KEYBOARD type step has two different forms as indicated above. Both forms indicate to the student that he is to type a response by printing three asterisks (***) on the bottom of the CRT. One of the forms of KEYBOARD must be used whenever you want the inquirer to respond to the text. KEYBOARD is also used to allow the inquirer to read text on the screen before it is erased. Even if no response is required, occasionally pauses are necessary. When KEYBOARD is used to allow inquirers time to read, he can indicate that he is finished by pushing SEND BLOCK without typing anything else.

Besides these other functions, the KEYBOARD:stepno form allows you to specify the step to which an inquirer typing @HELP will be sent. You remember that inquirers can always type a command such as @HELP instead of answering the question. (see Figure 1, page 5) Inquirers use @HELP when they are confused, so the @HELP step normally rephrases, defines vocabulary or provides additional information. If the inquirer uses @HELP at a KEYBOARD without a specified @HELP branch (KEYBOARD:stepno), he sees the message 'NO HINT PROVIDED, TRY PRESSING SEND BLOCK' and is given another chance to respond.

Part of a script using @HELP branches appears on page 14.

FIGURE 6

A @HELP SCRIPT FOR BEGINNERS

```

.
.
.
100.00 CRT WOULD YOU RATHER GO TO SCHOOL IN THE NORTH, SOUTH,
        EAST, OR WEST?
120.00 KEYBOARD:200.00
130.00 CRT DO YOU PREFER PUBLIC OR PRIVATE SCHOOLS?
140.00 KEYBOARD:300.00
160.00 GOTO 400.00
200.00 CRT WE CAN LIST THE SCHOOLS IN WHICHEVER AREA YOU WOULD
        LIKE. WHEN YOU HAVE SEEN ALL YOU WANT, USE @HELP
        AGAIN. WHICH AREA WOULD YOU LIKE LISTED?
        NORTH
        SOUTH
        EAST
        WEST
210.00 KEYBOARD:100.00
220.00 DECOMP:NORTH,SOUTH,EAST,WEST:230.00:240.00:250.00:260.00
230.00 CRT THE STATES OF THE NORTH ARE MONTANA, WISCONSIN, ...
235.00 GOTO 200.00
240.00 CRT THE STATES OF THE SOUTH ARE ALABAMA, MISSISSIPPI, ...
245.00 GOTO 200.00
250.00 CRT THE STATES OF THE EAST ARE MASSACHUSETTS, NEW YORK, ...
255.00 GOTO 200.00
260.00 CRT THE STATES OF THE WEST ARE CALIFORNIA, NEVADA, ...
265.00 GOTO 200.00
300.00 CRT PUBLIC SCHOOLS ARE THOSE RUN BY STATES AND CITIES.
        PRIVATE SCHOOLS OFTEN HAVE A RELIGIOUS AFFILIATION.
305.00 GOTO 130.00
.
.
.

```

Notes: Step 160.00 branches to 400.00 where the answer is processed.
Step 400.00 is not included here.

Section B2 KEYBOARD

Intermediate

KEYBOARD:stepno

Inquirers find the 'NO HINT PROVIDED, TRY PRESSING SEND BLOCK' message very frustrating, and they soon stop using @HELP at all. This is unfortunate since @HELP is one of the most powerful tools for improving the quality of the scripts. One suggested procedure is to use the next stepnumber in the KEYBOARD:stepno form. Even though the material is inappropriate, this is better than the message. Better still, provide @HELP steps at all KEYBOARDS, even if it is only a routine to collect a statement about the inquirer's confusion. Such statements often provide just the clue needed to revise a step that is producing unexpected and undesirable results.

An example of a script section using @HELP branches in this way follows.

FIGURE 7

A @HELP SCRIPT FOR INTERMEDIATES

```
.
.
.
100.00 CRT DO YOU PREFER WOOLIEGLOBBERING OR MASOCHISSERING?
120.00 KEYBOARD:160.00
130.00 DECOMP:WOOL$,MAS$:300.00:400.00
140.00 CRT+ PLEASE TYPE WOOLIEBLOBBERING OR MASOCHISSERING.
        USE @HELP IF NEEDED.
150.00 GOTO 120.00
160.00 CRT THERE IS NO HELP HERE, BUT WE COULD PROVIDE SOME FOR
        LATER INQUIRERS USING THIS SCRIPT IF YOU CAN TELL US
        WHAT YOUR PROBLEM IS. PLEASE TYPE IN A FULL STATEMENT
        NOW.
170.00 KEYBOARD:140.00
190.00 GOTO 140.00
.
.
.
```

[Footnote (advanced): To be more elegant and useful, you could include a step 180.00 using the logging function: 180.00 #(LG:HELP WANTED**#(CK))]

To continue with this example a bit further, you might find out by examining the record of use of this script that the meanings of woolie-globbering and masochisssing are not universally understood. On the basis of that information, you could revise the script segment to include genuine help. In the process, do not forget that some people needing @HELP may not want definitions. One possible revision follows.

FIGURE 8

A BETTER @HELP SCRIPT FOR INTERMEDIATES

```

.
.
.
100.00 CRT DO YOU PREFER WOOLIEGLOBBING OR MASOCHISSING?
120.00 KEYBOARD:160.00
130.00 DECOMP:WOOL$,MAS$:300.00:400:00
140.00 CRT+ PLEASE TYPE WOOLIEGLOBBING OR MASOCHISSING.
        USE @HELP IF NEEDED.
150.00 GOTO 120.00
160.00 CRT PERHAPS YOU ARE HAVING TROUBLE UNDERSTANDING ONE OF
        THE WORDS WOOLIEGLOBBING OR MASOCHISSING. WHICH
        WORD IS GIVING YOU TROUBLE?
165.00 KEYBOARD:180.00
167.00 DECOMP:0 (*BOTH EACH),(NEG):170.00:180.00
168.00 DECOMP:WOOL$,MAS$:170.00:175.00
170.00 CRT+ WOOLIEGLOBBING MEANS PULLING WOOLEN FUZZ FROM A RUG.
171.00 DECOMP:0 (*BOTH EACH) 0:175.00
172.00 GOTO 140.00
175.00 CRT+ MASOCHISSING MEANS KISSING DRY ICE.
176.00 GOTO 140.00
180.00 CRT PERHAPS YOU HAVE SOME OTHER DIFFICULTY WE HAVEN'T
        ANTICIPATED. PLEASE DESCRIBE YOUR PROBLEM IN FULL SO
        OTHERS CAN BENEFIT WHEN WE REVISE THE SCRIPT.
185.00 KEYBOARD:140.00
187.00 CRT+ THANKS, THAT WILL HELP US IMPROVE THE SCRIPT.
188.00 GOTO 140.00
.
.
.

```

Section C1 CRT Text

Beginner

This statement type causes the text following it to be written on to the CRT screen for the inquirer to read. After each KEYBOARD the text is erased and new text begins at the top of the screen. On any CRT screen, there will be a limit on the total number of characters on the screen at any one time. You should find out what the limit is before writing your script.

Remember that even if no response is required, you can use a KEYBOARD to allow the inquirer time to read and that will cause the screen to clear before the next CRT text is displayed. Therefore no matter how long your text, it can be displayed by breaking it into several parts with KEYBOARD statements. We have found that text longer than a hundred words is confusing and tends to bore inquirers.

Section C2 Other CRT Statements

Intermediate

There are several different CRT functions; CRT, CRT+, and CRTC. Each causes the computer to put text on the CRT for the inquirer to read. The different types control exactly where on the screen the text is placed and what happens to text already on the screen.

In general, CRT causes the screen to be cleared if the student has already had a chance to read the text on the screen because of a previous KEYBOARD. If he has not, CRT acts just as CRT+ does. After the screen is cleared, the new text begins at the top left of the screen.

CRT+ causes the text to be added to what is already on the screen beginning on the next available line below the old text. The CRTC statement type causes the text to be added to the screen in the next available space. The second 'C' in CRTC stands for continuous since the text then appears continuous on the screen. A flow chart of the exact method of deciding when to clear the screen is included in Appendix B.

Let us consider a brief script section to illustrate each of these types of CRT statements. Figure 10 [p. 19] shows how the screen would look after each step.

FIGURE 9

AN INTERMEDIATE CRT SCRIPT

```
.  
.   
.   
100.00 CRT HI THERE  
110.00 CRT+ NOW THEN LET'S GET TO WORK  
120.00 CRTC SINCE IT IS ABOUT TIME.  
130.00 KEYBOARD  
140.00 CRT YOU SAID SOMETHING THEN  
150.00 CRT THAT'S OK  
.   
.   
. 
```


after 100.00

HI THERE

after 110.00

HI THERE
NOW LET'S GET TO
WORK

after 120.00

HI THERE
NOW LET'S GET TO
WORK SINCE IT IS
ABOUT TIME

after 130.00

HI THERE
NOW LET'S GET TO
WORK SINCE IT IS
ABOUT TIME

after 140.00

YOU SAID SOMETHING
THEN

after 150.00

YOU SAID SOMETHING
THEN
THAT'S OK

FIGURE 10

THE SCREEN FOR FIGURE 9

In all CRT statements, the machine automatically inserts carriage returns at the ends of lines. New lines always begin after a full word, unless there are no spaces in the line. Individual screens will vary in the number of characters per line, but there will always be some limit. You need not be concerned with this, but you may want to in order to improve the appearance of the screen in list formats or multiple choice questions.

A number of formatting characters are available to you. You may write them into your script as you do other letters or symbols. The most commonly used is the carriage return $\begin{smallmatrix} C \\ R \end{smallmatrix}$. When it is placed in the text, it causes the next character to begin down one line and on the left margin of the screen.

Vertical tab $\begin{smallmatrix} V \\ T \end{smallmatrix}$ causes the next character to appear four lines lower and on the left margin of the screen. It can be used to separate the questions from the body of the text or for other display purposes. The horizontal tab $\begin{smallmatrix} H \\ T \end{smallmatrix}$ causes the next character to be displayed four positions to the right of the last character. The horizontal tab is useful for spacing and for emphasis.

On the Sanders 720 CRT, there is a blinking feature which can also be used to emphasize text. The relevant formatting symbols are 'start blink' $\begin{smallmatrix} S \\ B \end{smallmatrix}$ and 'clear blink' $\begin{smallmatrix} C \\ B \end{smallmatrix}$. These symbols cause any text between them to flash on and off while being displayed. Other CRT units may not have this feature.

When you are using formatting characters in your CRT statement you should be aware that there are only a limited number of lines and spaces on the CRT. By using too many $\begin{smallmatrix} C \\ R \end{smallmatrix}$'s and $\begin{smallmatrix} V \\ T \end{smallmatrix}$'s, you can exceed the space. In the ISVD implementation of GLURP on a Sanders 720 CRT, there are 768 character positions on a total of up to 23 lines. Other equipment will change the specific numbers involved.

Section C3 Formatting the CRT

Advanced

A number of formatting characters can be used to advantage on the Sanders 720 CRT in addition to the simple ones described before. Other equipment may not have these features. Complete description of the Sanders 720 is available from Sanders Associates, Inc.*

Among the features which found application in the ISVD are

Deleting a text block

Deleting a text line

Superimposing text on earlier text

Inserting text between other text

Multiple block structure

Type mode

To use any of these properly, you must understand how they work as explained in the Sanders 720 manual; you must introduce enough null characters to allow for timing on your particular installation; and finally, you must know the structure of the screen introduced automatically by GLURP. The first two are adequately explained in the Sanders manual. The explanation will not be produced here, since most installations will have other devices.

Assuming that you know all the Sanders terminology, the structure of the screen can be described here. There are two blocks, the first for inquirers (the KEYBOARD block) and the second for authors. Block one begins with six $\begin{smallmatrix} V \\ T \end{smallmatrix}$'s, and so appears near the bottom of the display area. It is nonetheless logically first. The KEYBOARD block is formatted for the inquirer and is used only in the 'type mode' so that the formatting can not be altered by the inquirer.

The second block is unformatted and used in the 'format type' mode by the system in accord with author instruction. GLURP automatically takes this structure into account. You must too if you intend to do anything fancy. The codes used for each formatting function and the multiple punches involved on the ISVD system are reproduced in Appendix C for your convenience.

* USER'S SYSTEM DESCRIPTION MANUAL for MODEL 720 COMMUNICATOR SYSTEM. 1966. Nashua, New Hampshire.

Section D1 Basic DECOMP

Beginner

DECOMP:rule:stepno

a. The Exact Match

The DECOMP statement is used to analyze an inquirer's response. DECOMP allows the computer to 'understand' what the inquirer has said. The computer does not know meanings for words; it simply matches letters according to your directions in the script DECOMP statements. If a match is correct, the computer branches as directed. The only knowledge of language the computer contains is what you build in through DECOMP.

Quite elaborate patterns of letters can be matched through DECOMP. While matching itself is a simple process, sophisticated results are possible. Proper use of DECOMP requires careful thought about language and understanding of the script context. If you put in the effort, your script will seem to understand. When the computer makes a foolish or inappropriate response to an inquirer, you know that the DECOMP statements were not given enough attention by the author. Let us begin simply and develop the full power of the DECOMP statement in steps.

DECOMP statements work on the response from the inquirer. Presumably some message from the computer has prompted him to reply, and you (through your script) want to know what he said so the computer can take appropriate action. Consider the following script segment as an example:

FIGURE 11

BEGINNER'S DECOMP SCRIPT

```
.  
. .  
. .  
050.00 CRT 21 IS THE SUCCESSOR OF 20. WHAT IS THE SUCCESSOR OF 8?  
060.00 KEYBOARD:075.00  
070.00 DECOMP:9:100.00  
075.00 CRT THE ANSWER IS 9. THE SUCCESSOR OF 8 IS 9.  
100.00 CRT WHICH OF THESE WOULD MOST ACCURATELY DESCRIBE YOU?  
A. I NEED A NEW EXAMPLE  
B. I WANT TO SEE THE LAST EXAMPLE AGAIN  
C. I GET IT. SKIP TO SOMETHING NEW
```



```

110.00  KEYBOARD:150.00
120.00  DECOMP:A:170.00
130.00  DECOMP:B:050.00
140.00  DECOMP:C:300.00
150.00  CRT  PLEASE TYPE A, B, OR C.
160.00  GOTO 110.00

```

```

.
.
.

```

This script segment begins with an example and a sample problem to teach the concept of 'successor'. Step 060.00 gives inquirers a chance to respond. Step 070.00 is the first DECOMP. The statement means 'see if the response is exactly '9'. If so, go to 100.00. If not, continue to the next step.' If the inquirer has not answered correctly, he is told the answer and then gets to step 100.00, which displays its text on the CRT. Step 110.00 allows the inquirer to respond. If he does not understand, and types @HELP, he will see the text at 150.00. He would then return to the KEYBOARD at 110.00. Suppose inquirer wants to review and therefore types 'B'. The DECOMP at 120.00 instructs the computer to see if 'A' matches the inquirer response. It does not, so the next step is 130.00, another DECOMP. This step asks if the 'B' matches the input. It does in this case, so the next step is 050.00. That step reviews the last example presented.

Several crucial features of DECOMP are illustrated in this example. First, the rule between colons (:) is matched against the inquirer's last input response. If the match does not work, the next step is taken as usual. If the match does work, the next step is the one specified in the DECOMP statement. DECOMP statements can test matches on more than single characters. A statement like

```
DECOMP:REVIEW:124.97
```

would test for the full word 'REVIEW'. The match would succeed on the response 'REVIEW', but fail on 'I WANT TO REVIEW' since the match is not exact in the second case. More than one word can be matched as in

```
DECOMP:I DO NOT KNOW:345.67
```

Notice that only an exact match will work. The response, 'WELL, I DO NOT KNOW YET' would fail to match this rule. Later we will see how GLURP helps you solve this problem.

b. Multiple Rules

Beginner

In the sample script (Figure 11 on page 22), three DECOMP steps (120.00, 130.00, and 140.00) are used to test for A, B, and C. Writing many steps is troublesome, so GLURP allows you to combine several tests in one statement:

```
DECOMP:A,B,C:170.00:050.00:300.00
```

This single statement is logically equivalent to the three statements in the sample script. First the inquirer's response is matched against A. If successful a branch to step 170.00 occurs. If not, a match against B is tried. If successful, the branch is to step 050.00. If not, the computer attempts to match against C. If successful on this last match, a branch to step 300.00 occurs. If not, the next step in the script is taken. Be sure that you understand that this one statement is exactly the same logically as the three original statements.

Let's take a moment to observe several important features of the syntax of the DECOMP statement. Every DECOMP statement begins with the seven characters 'DECOMP:'. Next, each rule up to the last is followed by a comma. The last rule is terminated by a colon. There are then as many step numbers as there are rules. The step numbers correspond left to right to the rules. The leftmost number is the branch for the leftmost rule. Step numbers are separated by colons. No colon follows the last step number. When the computer does a DECOMP, the rules are matched against the inquirer response from left to right until a match succeeds and a branch is taken. Once a rule matches, no rules to its right are used. If no rule of a DECOMP matches the inquirer response, the computer proceeds to the next step.

To be certain that this is clear, let's try some examples. Suppose you have a script segment such as this:

FIGURE 12

A SCRIPT WITH MULTIPLE RULES IN DECOMP

```

.
.
.
100.00  KEYBOARD
110.00  DECOMP:YOW,HOW ARE YOU,THAT IS NICE,NO:789.00:452.00:
        498.30:395.25
120.00  DECOMP:YOW,WHY NOT,HUM:395.00:385.02:405.00
.
.
.

```

<u>Student response</u>	<u>branch taken</u>
WHY NOT	385.02
CERTAINLY	none
YOW	789.00 (not 395.00)
NO	395.25

One note in passing. The total number of characters in a DECOMP statement is restricted by the implementation of the language. Different implementations will have different limits. At the ISVD, no more than 256 characters can be included in any DECOMP step.

c. The Zero (0) in DECOMP

Beginner

According to what we know so far, matches must be exact in GLURP DECOMP statements. That restriction needlessly limits the utility of DECOMP, so there is a way to get around it. Even in a multiple choice question, inquirers respond in strange ways:

```

I'LL TRY A
A SEEMS BEST
I GUESS A FOR ME
A FOR HEAVEN'S SAKE
I'D LIKE A NEW EXAMPLE

```

In our last sample script (Figure 11 on page 22), all these clearly indicate a branch to step 170.00, the new example. Unfortunately, the statement

DECOMP:A:170.00

would not match them. We need a way to tell the computer to ignore words before and after the word that interests us and complete the match anyway. The statement

DECOMP:0 A 0:170.00

does just that. The rule means: If the response has any number of words (including no words) followed by 'A' followed in turn by any number of words (including no words), this rule matches. Let's look in detail at each response to see how the matches work. In some cases both zeros (0) have words to match them. In others, only one has a match. In the first case, neither zero (0) has a match.

0	A	0
-----	A	-----
I'LL TRY	A	-----
-----	A	SEEMS BEST
I GUESS	A	FOR ME
-----	A	FOR HEAVEN'S SAKE
I'D LIKE	A	NEW EXAMPLE

It is easy to use zeros (0) to generalize your matching rules; sometimes too easy. Consider the input 'I HAD BETTER HAVE A REVIEW.' The correct interpretation (in the context of the script segment Figure 12 on page 22) is to branch to 050.00, but the new generalized rule

0 A 0

would match this input and give a new example. When using words rather than letters, it is easier to err by being too general. In an ISVD script about colleges, the statement

DECOMP:0 BOARD 0:295.00

sent an inquirer to material about testing and college boards. Unfortunately, many inquirers using the word 'BOARD' were asking about the cost of living at college (as in 'room and board'). The only way to avoid such problems is to be aware of them. If you have thought of both meanings of 'BOARD'

in a college context, you can use DECOMP to select the appropriate meaning from the rest of the input. If not, you are lost. Worse, the inquirer is confused.

d. The Synonym List

Beginner

Often you will want to recognize several synonyms in DECOMP matches. To establish that an inquirer is talking about schools, you could look for any of the following: SCHOOL COLLEGE UNIVERSITY. Easy enough.

DECOMP:0 COLLEGE 0,0 SCHOOL 0,0 UNIVERSITY 0:100.00:100.00:100.00

Any of the words would cause a branch to 100.00. They are all synonyms and are treated accordingly. Although tedious and inelegant, separate rules do work. More economically, but equivalently, you could write DECOMP:0 (*SCHOOL COLLEGE UNIVERSITY) 0:100.00

The computer interprets this as: if the input has any number of words (including no words) followed by any one of the words in parentheses followed by any number of other words (including no words), then this rule matches.

Notice the syntax of the synonym list: the words considered synonyms are enclosed in parentheses, separated by spaces. The first character in the parentheses is an asterisk (*). The asterisk is very important since it says to interpret the words that follow as synonyms. (Later you will learn about other uses of parentheses in DECOMP.) Several matches with this rule are given below.

0	(*SCHOOL COLLEGE UNIVERSITY)	0
I WANT TO GO TO	SCHOOL	-----
TELL ME ABOUT A GOOD	COLLEGE	FOR ME
WHAT	UNIVERSITY	HAS DRAMA
IS	COLLEGE	OR UNIVERSITY BETTER
THE ELECTORAL	COLLEGE	VOTED TODAY
A LARGE	SCHOOL	OF FISH SWAM BY

e. The Dollar Sign (\$) for Suffixes

Beginner

Often a class of words with the same root as TEACH TEACHER TEACHING TEACHED TEACHERS TEACHES is synonymous or equivalent in a rule. With what you already know, you could write

DECOMP:(*TEACH TEACHER TEACHING TEACHED TEACHERS TEACHES) 0:100.00

GLURP offers a more convenient notation called the 'dollar notation' designed to simplify such statements

DECOMP:TEACH\$ 0:100.00

The computer interprets the dollar notation to mean: if the input contains the letters 'teach' followed by any number of letters (including none) in the same word followed by any number of words, then this rule matches. The dollar sign must follow the word immediately with no intervening spaces. Some matches are illustrated below.

TEACH\$	0
TEACHING	LATIN IS TOUGH
TEACH	ME ABOUT COLLEGE
TEACHERISDUMB	TO USE THIS RULE FORM
TEACHRXZMFLBRTY	-----

The dollar notation conveniently allows you to match correct meaning in spite of misspelling of critical words. Examine the following script segment.

FIGURE 13

A SCRIPT USING DOLLAR SUFFIXES

```

.
.
.
100.00 CRT WHO WAS THE FIRST PRESIDENT OF THE UNITED STATES?
110.00 KEYBOARD:160.00
120.00 DECOMP:GEORGE WASHINGTON,G$ WASHINGTON:180.00:200.00
140.00 DECOMP:G$ WASH$,WASH$:220.00:240.00
160.00 CRT THE FIRST PRESIDENT WAS GEORGE WASHINGTON

```

```
170.00 GOTO 300.00
180.00 CRT CORRECT
190.00 GOTO 300.00
200.00 CRT THAT FIRST NAME IS SPELLED GEORGE.
210.00 GOTO 300.00
220.00 CRT RIGHT, BUT NOTICE THE SPELLING GEORGE WASHINGTON.
230.00 GOTO 300.00
240.00 CRT YES, WASHINGTON. NEXT TIME SPELL IT CORRECTLY
      AND USE HIS FIRST NAME, GEORGE.
```

.
.
.
.

Like all the ways of increasing the generality of a DECOMP rule, the dollar notation can cause trouble if used thoughtlessly. In an effort to catch the common misspellings of 'COLLEGE' (COLEGE COLLAGE COLLEJE COLLEDGE), one author used the rule

```
DECOMP:0 COL$ 0:234.60
```

While he caught all the common misspellings, he also caught words like COLLECT, COLORED, and COLUMN.

You may use the dollar (\$) notation on as many words in a rule as you like. Wherever you can write a word, you can use the dollar notation on that word. To revise a previous example, a more powerful rule would be

```
DECOMP:0 (*SCHOOL$ COLLEGE$ UNIVERSIT$) 0:100.00
```

Let us consider two examples illustrating what you now know about DECOMP statements.

FIGURE 14

CONTEXTUAL ANALYSIS WITH DECOMP

In talking to students about the cost of college, the word "resident" can mean either living in the state where the college is found or living at the college as opposed to commuting.

```
073.00 #(DS:PERSON:PERSON,STUDENT$,SOMEONE,PEOPLE)
      #(DS:PREP:COME$ FROM,LIVE$ IN)
074.00 #(DS:RESIDENT:(RESIDENT, STATE (PERSON),(PERSON) 0 (PREP) 0 STATE))
077.00 DECOMP:0 (RESIDENT) 0:400.00
.
.      (much omitted)
.
400.00 CRT THE COMPUTER HAS BEEN SMART ENOUGH TO PICK UP THE WORD RESIDENT
      FROM YOUR ANSWER, BUT CAN NOT TELL IF YOU ARE REFERRING TO SOME-
      ONE LIVING IN THE DORMITORY OR RESIDING IN THE SAME STATE AS THE
      COLLEGE HE IS CONSIDERING ATTENDING.
```

[note: This example is actually too advanced for this place.]

FIGURE 15

BRANCHES FOR DEEPER ANALYSIS

In the orientation scripts of the ISVD, each user statement is subjected to deeper and deeper analysis to locate the use within the script network. This process involves literally hundreds of DECOMP statements. A typical input would be subjected to the following matching DECOMPs. None of the failing DECOMP statements are included:

INPUT BY INQUIRER

TELL ME ABOUT THE COST OF TUITION AT COLLEGE

```
Match 1: 0 COLLEGE$ 0
Match 2: 0 (*COST FINANCING MONEY) 0
Match 3: 0 COST 0 TUITION 0
```

results in comments relating to the cost of tuition in college.

f. Phrases in DECOMP

Beginner

Phrases in certain contexts are equivalent to single words. For example, TRUE, OF COURSE, YES, and GUESS SO are all ways to say 'YES'. (at least in most contexts). With your present knowledge of DECOMP you might handle such a situation like this

Right DECOMP:0 (*YES TRUE) 0,0 OF COURSE 0,0 GUESS SO 0:100.00:100.00:100.00

Alternately you might write

Wrong DECOMP:0 (*YES TRUE OF COURSE GUESS SO) 0:100.00

The second statement is not the same as the first and is incorrect. Any one of the six words in the second statement will cause a match. In the first DECOMP, the phrases 'OF COURSE' and 'GUESS SO' are matched only if both words appear next to each other. If you have phrases that you want to consider equivalent, be sure that you use the first form.

g. External Lists of Phrases

Certain phrases are often used by all authors. GLURP provides a way to handle these easily. The method is called an external rule list. Some lists have been defined and are part of the system. Beginners in GLURP can use any of those defined for them. The lists include one called POS, which contains words and phrases meaning yes. The rule

DECOMP:(POS):100.00

will match any response that means 'YES'. The word 'POS' has nothing to do with the match. POS is the name of the list of YES words. The contents of the list includes words and phrases meaning YES. The POS list can be used in any script. The lists always available to you are these:

FIGURE 16

EXTERNAL LISTS OF PHRASES

POS	words and phrases meaning yes
NEG	words and phrases meaning no
UNSURE	words and phrases indicating uncertainty
QUEST	words indicating a question

DECOMP:(UNSURE),(NEG),(POS):100.00:150.00:200.00

DECOMP:(QUEST) (*COLLEGES\$ SCHOOL\$) 0:100.00

The first statement separates inquirers who are uncertain, negative, and positive in their responses. Whenever these lists are used together, they must be in the order above. The second example illustrates two distinct uses of parentheses in DECOMP. The parentheses around QUEST mean it is the name of a list, not a literal matching word. The parentheses around COLLEGES\$ SCHOOL\$ enclose an asterisk (*) which indicates a list of words present right in the statement.

The four rule lists POS, UNSURE, NEG, and QUEST are part of the GLURP language. The contents of the lists might be different in different installations. The words and phrases included in these lists at the ISVD are given in Appendix D.

You may define your own external lists in GLURP. Once defined, these behave just as those supplied for you. They will be available until deleted. To define them requires knowledge of weird functions at the intermediate level. Details are included below.

External lists may be either rule lists or word lists. Rule lists consist of phrases or words separated by commas. You must use a rule list form if your list contains phrases such as 'OF COURSE'. You save some processing time by using word lists where appropriate. Word lists contain items separated by spaces and treat items as synonyms like the internal word list identified by (*). Word lists should be defined externally when the same list will be used several times. If the list will be used only once, it is better to use the internal word list form: (*word word ...). To reference an external word list, you use the following notation:

```
DECOMP:(/WORDS):100.00
```

The slash (/) inside the parentheses indicates that the external list contains spaces as separators rather than commas.

Let us review the different kinds of lists in DECOMP.

```
DECOMP:0 (*COLLEGE$ SCHOOL$):100.00
DECOMP:0 (COLLEGE) 0:100.00
DECOMP:0 (/COLLEGE) 0:100.00
```

The first form, with the asterisk (*) as the first character inside the parentheses, indicates an internal word list. The words are displayed within the parentheses themselves and separated by spaces. The second form, with no special mark inside the parentheses, indicates an external word list. The list has to be defined somewhere before the DECOMP in question uses it, and has commas separating the rules. The third form has a slash (/) as the first character within the parentheses. It is called the external word list form, meaning that the list is defined elsewhere and has spaces separating the items on the list. Both external forms --- (/name) and (name) --- use the name of a list defined elsewhere. The name is not used in the matching itself. The internal word list --- (*word word word) --- contains the matched words themselves within the parentheses.

To create your own external word or rule list, you use the function Define String (DS). The format is

#(DS:N:(list))

N is the name of the list. For use in DECOMP, the name can contain no blanks, commas, colons, periods or question marks. It is better to use short names of three to six characters. The name N must exactly match that which appears in the DECOMP between parentheses

DECOMP:(N):100.00
DECOMP:(/N):100.00

list is either word or rule list depending on whether it includes phrases or just single words to be tested as synonymous.

You must be certain that you define a list before you are going to use it, and that you call it by its exact name. If you neglect to do so, you will get an error message: EXTERNAL LIST UNDEFINED.

Within external rule lists, each rule can be anything that is a rule in a DECOMP statement. It may in particular include references to other external rule and word lists as well as its own internal word lists.

Please note the use of parentheses in the Define String:

#(DS:N:(list))

The second set of parentheses protects any parentheses within the 'list' you write.

Let us consider a short piece of script:

FIGURE 17

SCRIPT WITH DECOMP LISTS

```
.
.
.
100.00 #(DS:SCHOOL:(SCHOOL$ UNIVERSIT$ COLLEGE$))
      #(DS:TEST:(COLLEGE BOARD$,(*SAT ACTP CEEB),ENTRANCE EXAM$))
120.00 CRT  WHAT CONCERNS YOU ABOUT COLLEGE?
130.00 KEYBOARD
140.00 DECOMP:0 (/SCHOOL) 0 (TEST) 0:200.00
.
.
.
```

Some sample responses at 130.00 are given below. Those that match are displayed under the part of the rule they match.

0	(/SCHOOL)	0	(TEST)	0
THE	COLLEGE	-----	ENTRANCE EXAMS	SCARE ME
-----	UNIVERSITIES	REQUIRE	COLLEGE BOARD	TESTS
AT MY	SCHOOL	WE DO NOT CRAM FOR	CEEB	TESTS
-----	SCHOOL	-----	ACTP	-----

failures:

I FAILED MY COLLEGE BOARDS
I'M WORRIED ABOUT SAT SCORES

no test word -- COLLEGE is the SCHOOL word
no SCHOOL word

Section D3 Tricky Details of DECOMP

a. The asterisk in external lists

There is one further feature of DECOMP statements. This provides flexibility and some compactness in external rule lists. On the ISVD system, the feature is nominally present, but not debugged. For each rule in an external rule list, you may choose a different branch. Thus one list provides multiple branches. An example of the notation is as follows:

```
DECOMP:(MULTI:105.00:133.00:*:138.00):129.00
```

NAME OF FORM	VALUE
MULTI	YES,MAYBE 0,I THINK SO 0,SURE\$ 0

Notice first the structure. The rule list is named immediately inside the parentheses. The colons also are inside the parentheses in contrast to the normal DECOMP form. Each step number inside parentheses corresponds to its subrule. In the example, 105.00 is taken as the branch if the answer is 'YES', 133.00 for 'MAYBE 0'. The asterisk (*) is used to indicate that the normal branch for that entire rule list (129.00) should be taken in case of a match with that subrule. In the example above, an equivalent form without the asterisk is

```
DECOMP:(MULTI:105.00:133.00:129.00:138.00)
```

The asterisk is convenient when most subrules go to the same place. If all go to the same step, we have our usual condensed notation.

```
DECOMP:(MULTI:133.00:133.00:133.00:152.00):
```

```
DECOMP:(MULTI:*:*:*:152.00):133.00
```

```
DECOMP:(MULTI:133.00:133.00:133.00:*):152.00
```

are all equivalent.

```
DECOMP:(MULTI:133.00:133.00:133.00:133.00):
```

```
DECOMP:(MULTI:*:*:*:*):133.00
```

```
DECOMP:(MULTI):133.00
```

are also equivalent, but the last form is clearly most economical. These subrule designations can be used in combination with other rules in the same DECOMP statement, but do not make sense in complex rules. The following examples illustrate correct and incorrect uses of the subrule convention.

Right DECOMP:(MULTI:***:120.00),(BAD):150.00:125.00
 Right DECOMP:(MULTI:***:125.00) 0 NOPE:133.00
 Wrong DECOMP:(MULTI:*:125.00:130.00:*) 0 (BAD:*:151.00):126.00

In the first example, three subrules of the first rule go to 150.00, one goes to 120.00. A match with (BAD) would go to 125.00. The second example is not ambiguous, and GLURP handles it. The third example is clearly wrong. Suppose that a response matched the second subrule of each external rule list. Should the branch go to 125.00 or 151.00 or compromise on 126.00?

b. Multiple Sentence Input

Advanced

The DECOMP statement can make some absurd matches if the inquirer types more than one sentence. A word in one sentence and a phrase in another could each be matched in the same rule. To avoid this, GLURP specifies that a match must take place entirely within one sentence. As an example, take the statement

DECOMP:HOW 0 NOW 0 COW:100.00

Response	Result
HOW ARE YOU NOW? YOU LOOK LIKE A COW.	fails
HOW NOT BROWN COW	matches
WHAT'S THAT? HOW ARE YOU NOW, COW.	matches

WEIRD FUNCTIONS IN GLURP

Section A1 Introduction

Beginner

GLURP contains some strange looking notation. It is full of #() and sometimes looks as bad as this:
#((#(DS:A:WOW))#(AN:#(N)#(PRE)):F#(M))
or even worse. While the full complexity of the notation and concepts are difficult, you can use the notation effectively without understanding everything implied by what you write.

Section A2 Slides

Beginner

The SLIDE statement provides an example of this notation. You can tell the computer to display a slide by writing:

`$(SLIDE:name:)`

The 'name' section represents any name of a slide. If the group you work with has a number of slides available for use in scripts, each will have a name. If you want your script to show one, just write the appropriate SLIDE statement with the selected name. If you want to use a slide not yet available at your installation, one you make up yourself, get help from the person in charge of visual displays.

Section A3 Print String

Beginner

You already know one way to print text using the CRT statement. There is another way using the `#( )` notation. To print text, you may write:
`$(PS:text)`

The notation is read "Print the string 'text' on the CRT." For most 'text' this is equivalent to

CRT text
and far less convenient. Why go to all the trouble of having two ways to do the same thing? Why introduce complex notation when there is a simple way to do it? Glad you asked. Certain texts demand the special notation. The reason lies in the special way in which `#( )` steps are treated. This will become clear as we introduce a few new statement types. Bear with it for a few paragraphs, and then you will understand why we need the `#( )` notation.

Section A4 Inquirer Characteristics

Beginner

Many different inquirers will use your script. Sometimes you will want to call the inquirer by his name rather than saying 'Hey you'. However, you have no way of knowing who will use your script when you are writing it. To solve this problem, you can use the NAME statement. You write #(NAME) and the computer replaces it by the name of the inquirer using the script at that time. To display the name on the screen, you need to use the Print String (PS) function:

```
$(PS:HELLO, $(NAME))
```

This would appear on the screen as

```
HELLO, PAT YEE           or
```

```
HELLO, MARIETTA HALEY   or
```

```
HELLO, NONI STRAUSS
```

or whatever the name of the current inquirer happened to be.

Suppose you wrote

```
CRT HELLO $(NAME).
```

The inquirer would see

```
HELLO $(NAME).
```

(and think that the author was cracking up). Statements like #(NAME) are meant to be "replaced by their value." This will only happen if the first character of the step is #. This is why we need two ways of obtaining displays on the CRT.

Notice that the #(NAME) statement is replaced by the full name of the inquirer. This often sounds formal and stuffy. To print the first name of the inquirer instead of the full name, you may write

```
$(PT:NAME)$(CR:NAME)
```

wherever you want the first name to appear. Use this as a formula without trying to understand. It is intermediate level weird, but so useful that it is included here for beginners.

Other statements give other characteristics of the inquirer. They behave just as #(NAME) does. That is, they are replaced by their values whenever they appear in a function type statement. Consider the statement: \$(PS:HELLO \$(NAME). YOU ARE \$(AGE) YEARS OLD IN \$(GRADE)TH GRADE)

When the inquirer sees it, it would say something like:

HELLO TOM HUTCHINSON. YOU ARE 15 YEARS OLD IN 5TH GRADE.

The #(SEX) function works similarly. #(SEX) is replaced by 'MALE' or 'FEMALE' depending on the sex of the inquirer.

To summarize the inquirer characteristics statements available to you, we have this table:

FIGURE 18

INQUIRER CHARACTERISTICS

FUNCTION	REPLACED BY
\$(NAME)	full name of the current inquirer
\$(PT:NAME) \$(CR:NAME)	first name of the current inquirer
\$(SEX)	'MALE' or 'FEMALE'
\$(GRADE)	grade in school as a number
\$(AGE)	age in years as a number

Section A5 Equals

Beginner

While the inquirer characteristics functions are useful when displayed directly to the inquirer in the form of text, they attain their greatest utility when they help discriminate among different kinds of inquirers. When used in this way, the inquirer characteristics functions allow you to talk about college choice with a junior or senior in the same script. Some material can be given to both, but some is relevant only to the senior who faces the choice immediately; other material is relevant only to someone with more time to spend getting ready for the decision. In a similar vein, examples about the world of work are often sex specific. You could write one script, but you should provide different examples depending on the sex of the inquirer.

The Equals function (EQ) makes all of this possible. In form, the Equals (EQ) function looks like this:

```
EQUALS    #(EQ:X1:X2:Z1:Z2)
```

Do not be put off by the funny symbols. X1 and X2 stand for any two pieces of text. If X1 and X2 are identical, the whole function is replaced by Z1. If X1 and X2 differ in any way, the whole function is replaced by Z2. This is a lot harder than the other functions, but it is worth it. An example might help.

```
$(PS:YOU GOT A NEW  
  $(EQ:$(SEX):MALE:MODEL AIRPLANE:DOLL HOUSE)  
  FOR YOUR BIRTHDAY.)
```

If a male were using the script when this step came up, he would see:
YOU GOT A NEW MODEL AIRPLANE FOR YOUR BIRTHDAY.

A female would see:

YOU GOT A NEW DOLL HOUSE FOR YOUR BIRTHDAY.

That is reasonable enough, but how did it happen. Let's examine it in detail. Suppose that a boy is using the script, so the value of \$(SEX) is 'MALE'. The computer looks at the step from left to right until it finds a function with a right parenthesis. In this case, two statements begin with \$((the PS and the EQ) but neither of them ends with a right parenthesis before the \$(SEX) function appears with its right parenthesis. The computer therefore does the \$(SEX) function first and replaces it by its value, leaving:

```

#(PS:YOU GOT A NEW
  #(EQ:MALE:MALE:MODEL AIRPLANE:DOLL HOUSE)
  FOR YOUR BIRTHDAY.)

```

The next right parenthesis the machine finds belongs to the Equals (EQ), so it does it. In terms of the description on the previous page, we have

X1 = MALE

X2 = MALE

Z1 = MODEL AIRPLANE

Z2 = DOLL HOUSE

According to the instructions, if X1 and X2 are identical (as in this case) the whole function is replaced by Z1. The step now reads:

```

#(PS:YOU GOT A NEW
  MODEL AIRPLANE
  FOR YOUR BIRTHDAY.)

```

The only remaining statement is the Print String (PS). It causes the text to be printed on the screen. Try to follow the same sequence yourself for a female inquirer. We will do some other examples when you have learned a few more functions.

In addition to text as the values of Z1 and Z2 in an Equals (EQ) statement, you can use a branch or GOTO notation.

*stepno

The computer interprets the asterisk to mean that the numbers are a branch, not regular text. In other words, the * means the same as GOTO. Don't use GOTO itself in this way. You could use the *stepno notation like this:

FIGURE 19

SCRIPT USING *stepno

```

.
.
.
100.00  #(EQ: #(GRADE):12:*110.00:*150.00)
110.00  CRT  IT IS QUITE LATE TO START PLANNING FOR COLLEGE NOW,
        BUT BETTER LATE THAN EARLY.
150.00  CRT  LET'S BEGIN THEN.
.
.
.

```

Section A6 Define String

Beginner

DEFINE STRING

\$(DS:N:X)

The Define String (DS) function creates a new function called a 'form' with the name N and value X. You create new functions in order to use them. If you use the function N late by writing \$(N), it will be replaced by X.

Suppose early in your script you, on the basis of some inquirer response, decided that the current inquirer was a DUMMY. You might write a statement like

\$(DS:TYPE:DUMMY)

Later, you could use it directly in a statement like

\$(PS:WE FOUND OUT EARLIER THAT YOU ARE A \$(TYPE).)

The inquirer would see

WE FOUND OUT EARLIER THAT YOU ARE A DUMMY.

At other places in the script you might make different evaluations of your inquirers:

\$(DS:TYPE:SMARTY)

\$(DS:TYPE:COCKY, OVERCONFIDENT STUDENT)

The inquirers would then see different text as a result. You can use Equal (EQ) together with your own functions to branch to appropriate material, just as we did with pre-determined functions like \$(SEX) on p. 43.

Suppose that at various places in your script you have defined TYPE as DUMMY, SMARTY, or COCKY, OVERCONFIDENT STUDENT. You can use these later to decide what to tell him.

FIGURE 20

USE OF AUTHOR DEFINED FUNCTIONS

```
.
.
.
060.00  #(EQ: #(TYPE):SMARTY:*200.00)
        #(EQ: #(TYPE):DUMMY:*100.00)
        #(EQ: #(TYPE):COCKY, OVERCONFIDENT STUDENT:*150.00)
100.00  CRT  LET'S GO BACK TO THE BASICS.  THE COUNTRY IS MADE UP
        OF STATES.  EACH STATE HAS A REPRESENTATIVE OR MORE
        IN THE HOUSE OF REPRESENTATIVES.
110.00  KEYBOARD
112.00  GOTO 300.00
150.00  CRT  IF YOU THINK YOU ARE SO SMART, WHAT WAS THE VOTE ON
        HOUSE BILL #1032 IN 1956?
160.00  KEYBOARD
170.00  CRT  NOT SO.  IT WAS 32 TO 239, SO THE BILL FAILED.  MAYBE
        YOU WON'T BE SO WISE NEXT TIME.
180.00  GOTO 300.00
200.00  CRT  YOU SEEM TO HAVE A BASIC UNDERSTANDING OF THE SEPARA-
        TION OF BRANCHES OF THE GOVERNMENT.  YOU CAN SKIP THE
        NEXT SECTION.
210.00  GOTO 400.00
.
.
.
```

Notice that the classification early in the script allows you to tailor the material to the individual in this script segment. In this way, the Define String (DS) function is very useful to you as an author.

Section A7 Arithmetic

Beginner

One particularly interesting kind of function has a number as value. Several arithmetic functions are available to manipulate numbers.

FUNCTION	NEW VALUE OF N	EXPLANATION
#(DS:N:1)	1	create string called N whose value is 1
#(AN:N:1)	2	adds one to the value of N
#(AN:N:5)	7	adds five to the value of N
#(AN:N: #(N))	14	doubles N (note difference between N and #(N))
#(SN:N:2)	12	subtracts two from N
#(SN:N: #(N))	0	makes N zero. $12 - 12 = 0$
#(SN:N:4)	-4	4 is subtracted from N. $0 - 4 = -4$
#(MN:N:3)	-12	multiplies N by three
#(MN:N: #(N))	+144	N is squared
#(DN:N:4)	36	becomes the largest whole number in the quotient of N and 4.

Any string name can be used. N is only an example. The value of the function named N and of the third argument (X) must be whole numbers. No decimals or fractions are allowed. Not only is the value of the function N changed, the function N is replaced by its new value inside the function in which it appears. This works just as #(TYPE) does in Figure 20.

Let's follow an example of arithmetic functions at work.

NEW VALUES			
FUNCTION	M	R	EXPLANATION
#(DS:M:5) #(DS:R:21)	5	21	establishes the functions
#(AN:M:3) #(DN:R:4)	8	5	$21 \div 4 = 5.25$ rounded down to 5 no fractions are allowed
#(SN:M:6)	2	5	$8 - 6 = 2$
#(MN:R: #(R))	2	25	becomes #(MN:R:5). $5 \times 5 = 25$
#(DN:R:3)	2	8	highest whole number in $25 \div 3$
#(MN:M: #(R))	16	8	becomes #(MN:M:8). $2 \times 8 = 16$
#(DN:R: #(SN:M:14))	2	4	becomes #(DN:R:2). M is defined as $16-14=2$ then redefined $8 \div 2 = 4$

FUNCTION	NEW VALUES		EXPLANATION
	M	R	
#(AN:R:-6)	2	-2	$4 - 6 = -2$
#(SN:M:-8)	10	-2	$2 - (-8) = 2 + 8 = 10$
#(MN:R:-#(AN:M:-2))	8	16	becomes #(MN:R:-8). M is defined as $10 - 2 = 8$. Then $R = -2 \times -8 = 16$

Section A8 Compare

Beginner

A special function is used to compare numbers. It is called Compare (CP) and is written:

COMPARE

\$(CP:D1:D2:Z1:Z2:Z3)

D1 and D2 are numbers

Z1, Z2 and Z3 are functions, phrases, or branches (*stepno)

When a Compare function is recognized, the values of D1 and D2 are compared.

If D1 is less than D2, Z1 replaces the whole function.

If D1 and D2 are equal, Z2 replaces the whole function.

If D1 is greater than D2, Z3 replaces the whole function.

As an example,

\$(CP:\$(M):5:*200.00:*210.00:*220.00)

For \$(M) =	Branch Taken
3	200.00
5	210.00
7	220.00
undefined	200.00 (understood as 0 in Compare)
-21	200.00
WOW	error in arithmetic statement

Suppose you have counted errors throughout a script using \$(AN:ERROR:1). One use of the Compare (CP) function might be the step \$(PS:YOU HAVE

\$(CP:\$(ERROR):5:LESS THAN:EXACTLY:MORE THAN)

FIVE ERRORS.)

Notice that it would be an error to write

\$(CP:ERROR:5:LESS THAN:EXACTLY:MORE THAN)

since the letters ERROR are not a number. You must cause GLURP to evaluate the contents of a function you or someone has named ERROR, e.g. by \$(DS:ERROR:2). You cause GLURP to evaluate this by writing \$(ERROR).

Section A9 Inquirer Response

Beginner

Two other functions that deserve particular attention by the beginning GLURP'er allow you to use the inquirer's responses directly to personalize the text. These are the Call Keyboard (CK) and the Keyboard Segment (KS) functions.

CALL KEYBOARD #(CK)

The function Call Keyboard (CK) is replaced by the inquirer's last response. You can use it to save responses in a function for later reference.

#(DS:LAST:#(CK))

Then you could use this function after a while to say

#(PS:BEFORE YOU SAID #(LAST))

Notice that writing

#(PS:BEFORE YOU SAID #(CK))

is not the same since the #(CK) always refers to the current response. You could also use the function in a branching text:

#(EQ:#(CK):5:*120.00:*130.00)

KEYBOARD SEGMENT #(KS:D)

D is a decimal number. The function is replaced by the Keyboard Segment D from the most recent DECOMP match. The Keyboard Segment (KS) function relates to the last DECOMP rule matched since the latest KEYBOARD. If there has been no match since the last KEYBOARD, Keyboard Segment is meaningless. When a rule matches in DECOMP, it creates 'Keyboard Segments' that correspond to the parts of the rule.

As an example, consider the rule

0 DOES A\$ 0 (*DO MAKE)

which has 4 parts: 0, DOES A\$, 0, and (*DO MAKE).

One possible matching response is

'HOW MUCH DOES AN ARC WELDER MAKE'

KEYBOARD SEGMENT	MATCH	RULE PART
1	HOW MUCH	0
2	DOES AN	DOES A\$
3	ARC WELDER	0
4	MAKE	(*DO MAKE)

Another match would be the response

'WHAT KIND OF WORK DOES AN ABRASIVE GRADER DO'

It would create the Keyboard Segments

KEYBOARD SEGMENT	MATCH	RULE PART
1	WHAT KIND OF WORK	Ø
2	DOES AN	DOES A\$
3	ABRASIVE GRADER	Ø
4	DO	(*DO MAKE)

You can use the Keyboard Segment (KS) function to personalize the text your script displays. Let's use the example DECOMP rule above to demonstrate this ability in a short script segment.

FIGURE 21

SCRIPT USING KEYBOARD SEGMENT

```

.
.
.
100.00 CRT WHAT WOULD YOU LIKE TO KNOW ABOUT JOBS?
110.00 KEYBOARD
300.00 DECOMP:0 DOES A$ 0 (*DO MAKE):425.00
425.00 #(PS:ARE YOU CONSIDERING BECOMING A #(KS:3)? IS THAT WHY
      YOU WANT TO KNOW #(KS:1) THEY #(KS:4)?)
.
.
.

```

With our two sample responses analyzed before, step 425.00 would say
 ARE YOU CONSIDERING BECOMING A ARC WELDER? IS THAT WHY YOU WANT
 TO KNOW HOW MUCH THEY MAKE?

or

ARE YOU CONSIDERING BECOMING A ABRASIVE GRADER? IS THAT WHY YOU
 WANT TO KNOW WHAT KIND OF WORK THEY DO?

Section A10 Miscellaneous Other Useful Information

Beginner

Other functions that beginners can use without fear are listed below. Beyond those, lies the intermediate area. With other functions, you must know more about how GLURP actually works.

NUMBER #(NUMBER:X:STEPNO1:STEPNO2)

This function determines if the string X is a number or not. If so, a branch to STEPNO1 occurs. If not, the branch is to STEPNO2.

TAKEN (TAKEN:D:STEPNO1:STEPNO2)

This function tests if an inquirer has seen script numbered D. If so, a branch to STEPNO1 results. If not, the branch is to STEPNO2. See your group leader for a list of scriptnames and numbers.

AORAN #(AORAN:X)

This function is replaced an AN X if the string X begins with a vowel. Otherwise, it is replaced by A X.

Finally, there are some important things for beginners to know and use. Understanding requires knowledge at the intermediate or advanced levels.

1.) Try to number steps by tens. 100.00, 110.00, 120.00, etc. This is to give you plenty of space to insert other steps later to elaborate or to correct your script.

2.) One of your first steps should be

#(NT:CLEANUP:(#(DD:N1:N2:...)))

N1, N2, N3,... are the names of strings you define using #(DS:N1:X) elsewhere in the script.

3.) When your script ends, write a step like this:

#(CLEANUP)#(XQ: #(PT:LINE): #(PT:LINK))

4.) Your script will have a number assigned to it by the group leader. Suppose it is 132. Write a step numbered 000.97

000.97 #(TO:STAKEN:132)

BEGINNERS CAN STOP HERE.

Section B1 Intermediate Weird Introduction

Intermediate

a. Nesting Functions

At the intermediate level of GLURP, you can use forms storage, manipulate forms pointers, store and retrieve data, and combine binary vectors. All that gibberish will be explained as we get to it. First, there are several notions underlying all intermediate use of GLURP. These are using functions in place of strings of characters, using protective parentheses, and using form names and form values.

GLURP allows you to use any string of characters including other functions as part of the function. This idea is illustrated by a function like:

`#(EQ: #(N): 5: *102.00: *213.00)`

The function `#(N)` is included in the Equals (EQ) function. You are already familiar with this simple use of functions within other functions. More generally, any place you can write a string of characters in a function, you can write another function as well. If you can write

`#(DS: NAME STRING: CONTENTS)`

you can as legally write

`#(DS: #(NAME) #(S): #(CAT) #(SEX))`

or even

`#((#(A) #(B): #(C) #(D): #((#(A) B #(C)) #((#(D) #(E)): #(B)))`

(whatever that means).

You already know the Define String (DS) function from beginning GLURP. You remember that it creates a function with a given name and contents. Functions created by script authors are called Forms and are kept in Forms Storage. The function

`#(DS: CAT: MEOW)`

creates a form in forms storage whose name is CAT and whose value (or contents) is MEOW. Think of forms storage as a table of forms like this:

<u>NAME OF FORM</u>	<u>VALUE</u>
CAT	MEOW
NAME	SUSAN TREMBLE
SEX	FEMALE
GRADE	12
AGE	17

You can add new forms by using Define String (DS). You can use forms by asking for their values.

```

#(CP:21: #(AGE):*129.00:*139.22:*302.56)
#(PS: #(EQ: #(SEX):MALE: #(AGE): #(CAT)))

```

After the following functions execute the table of forms will have the modified values given below:

```

#(DS:MEOW:SCRATCH)
#(DS:PET:CAT)
#(DS:CHILD:PET)

```

<u>NAME OF FORM</u>	<u>VALUE</u>
CAT	MEOW
MEOW	SCRATCH
PET	CAT
CHILD	PET
NAME	SUSAN TREMBLE
SEX	FEMALE
GRADE	12
AGE	17

Now we can write some functions and their values:

<u>FUNCTION</u>	<u>VALUE</u>
#(PET)	CAT
#(MEOW)	SCRATCH
#(SCRATCH)	nothing -- no such form exists
#(#(PET))	MEOW occurs in two stages. After the first it says #(CAT)
#(#(#(PET)))	SCRATCH occurs in three stages.

b. Protective Parentheses

Intermediate

Now suppose you want to define a form whose value is #(CAT). (You often will want to do similar things as we will see in a moment.) You could write

Wrong `#(DS:OOPS: #(CAT))`

This would create a new form with name OOPS and value MEOW. This is not what you wanted. It occurs because the innermost function is evaluated first.

<code>#(DS:OOPS#(CAT))</code>	original
<code>#(DS:OOPS:MEOW)</code>	evaluate #(CAT)

You could try to solve the problem with

Wrong `#(DS:OOPS: #(#(CAT)))`

but you get a form OOPS with contents SCRATCH. This process takes two steps instead of one.

GLURP offers a way around this problem. You can protect a function and keep it from being evaluated by surrounding it with parentheses. Parentheses used in this way are called protective parentheses.

`$(DS:OOPS:($(CAT)))`

creates a forms named OOPS whose contents is `$(CAT)`. Protective parentheses are a fundamental concept in GLURP, and the source of much of its power. They are also illusive and hard to grasp. Examples to help you understand follow.

The following functions create the table of forms below.

`$(DS:A:B)$(DS:B:$(A))`
`$(DS:C:$(B))`
`$(DS:D:$(C))`
`$(DS:E:$(C))`

NAME OF FORM	VALUE
A	B
B	B
C	\$(B)
D	\$(C)
E	B

The last instance is the hardest. Examine the stages of processing:

`$(DS:E:$(C))` original
`$(DS:E:$(B))` evaluate \$(C)
`$(DS:E:B)` evaluate \$(B)

Now continue with a few more

`$(DS:BB$(B):$(A))`
`$(DS:AA$(A):$(D))`
`$(DS:D$(D))$(D):$(B))`

NAME OF FORM	VALUE
A	B
B	B
C	\$(B)
D	\$(C)
E	B
BBB	\$(A)
AAB	\$(D)
D\$(D)B	B

The last function is most complex, so let's trace it:

`$(DS:D$(D))$(D):$(B))` original
`$(D):$(D)$(D):$(B))` remove protection

#(DS:D#(D)#(C):#(B))	evaluate #(D)
#(DS:D#(D)#(B):#(B))	evaluate #(C)
#(DS:D#(D)B:#(B))	evaluate #(B)
#(DS:D#(D)B:B)	evaluate #(B)

When a set of protective parentheses are encountered, they are removed. Anything inside of the protective parentheses is protected. It is not evaluated or changed in any way. That sounds easy enough. Try it on this function.

#(DS:BB:((#(A))))

The protective parentheses are these

#(DS:BB:((#(A))))

They are lost, to create the next stage

#(DS:BB:(#(A)))

No further evaluation takes place in there, so the form BB has value (#(AA)).

A more complex example is the function

#(DS:(#(A))#(B):#(A)(#(A))#(BB))

The evaluation steps are as follows:

#(DS:#(A)#(B):#(A)(#(A))#(BB))	drop protective parentheses
#(DS:#(A)B:#(A)(#(A))#(BB))	evaluate #(B)
#(DS:#(A)B:B#(A)(#(A))#(BB))	evaluate next #(A)
#(DS:#(A)B:B#(A)#(BB))	drop protective parentheses
#(DS:#(A)B:B#(A)(#(A)))	evaluate #(BB)
#(DS:#(A)B:B#(A)#(A))	drop protective parentheses

The form #(A)B has value B#(A)#(A)

Try this one yourself before looking at the answer on the next page.

#(DS:#(#(A)#(B))(#(A)):((#(A)#(B)))#(B))

The form $\#(A)\#(A)$ has value $(\#(A)\#(B))B$

If you got a different answer, follow the step by step trace below.

$\#(DS:\#(\#(A)\#(B))(\#(A)):(\#(A)\#(B)))\#(B))$	original
$\#(DS:\#(B\#(B))(\#(A)):(\#(A)\#(B)))\#(B))$	evaluate $\#(A)$
$\#(DS:\#(BB)(\#(A)):(\#(A)\#(B)))\#(B))$	A evaluate $\#(B)$
$\#(DS:(\#(A))(\#(A)):(\#(A)\#(B)))\#(B))$	evaluate $\#(BB)$
$\#(DS:\#(A)(\#(A)):(\#(A)\#(B)))\#(B))$	drop protective parentheses
$\#(DS:\#(A)\#(A)):(\#(A)\#(B)))\#(B))$	drop protective parentheses
$\#(DS:\#(A)\#(A):(\#(A)\#(B))\#(B))$	drop protective parentheses
$\#(DS:\#(A)\#(A):(\#(A)\#(B))B)$	evaluate $\#(B)$

Most of the time we use protective parentheses in far less complex statements such as this:

$\#(CP:\#(N):5:(\#(DS:M:LOW)):(\#(DS:M:EQUAL)):(\#(DS:M:HIG)))$

Here the form named M will have one value, that which describes the relationship between the value of N and 5. Notice that this function is not the same as:

$\#(CP:\#(N):5:\#(DS:M:LOW):\#(DS:M:EQUAL):\#(DS:M:HIG))$

The latter statement always leaves the value of 'M' as 'HIGH'. Each of the Define Strings (DS) works in turn as it is encountered, leaving the value of M as the last one executed. The comparison (CP) function is irrelevant. With the first form, the action is like this:

$\#(CP:3:5:(\#(DS:M:LOW)):(\#(DS:M:EQUAL)):(\#(DS:M:HIG)))$	
$\#(CP:3:5:\#(DS:M:LOW):(\#(DS:M:EQUAL)):(\#(DS:M:HIG)))$	drop protective paren.
$\#(CP:3:5:\#(DS:M:LOW):\#(DS:M:EQUAL):(\#(DS:M:HIG)))$	drop protective paren.
$\#(CP:3:5:\#(DS:M:LOW):\#(DS:M:EQUAL):\#(DS:M:HIG))$	drop protective paren.
$\#(DS:M:LOW)$	take appropriate value

c. Names and Values in Arithmetic Functions

Intermediate

Another concept crucial to intermediate level GLURP is the difference between the name of a form and the value of that form. Functions like Define String (DS) and Equals (EQ) require either a form name or a string. You must be clear which type is wanted. To illustrate this point, let us consider the eight arithmetic functions. You already know the four using a form name: (AN), (SN), (DN), and (MN). There are four other arithmetic functions requiring a decimal number instead of a name. They are

ADD	#(AD:D1:D2)
SUBTRACT	#(SU:D1:D2)
DIVIDE	#(DV:D1:D2)
MULTIPLY	#(ML:D1:D2)

Contrast the two different addition functions. The Add to Form Name (AN) requires a name as its second argument. Add (AD) requires a number as its second argument. Their actions are different, but they are easily confused unless understood. The Add (AD) function sums the two numbers D1 and D2. The sum replaces the function wherever it appears, e.g. #(PS: #(AD:2:3)) causes 5 to be displayed on the CRT. Nothing happens to forms storage. The Add to Form Name (AN) requires a form name N. The value of the form N is added to D. The result becomes the new value of form N. The sum also replaces the original function, e.g. #(DS:N:5) #(PS: #(N)) = 5 prints #(PS: #(AN:N:2)) = 7 prints #(PS: #(N)) = 7 prints.

Suppose you had forms storage as follows. Notice that a name of a form can be a number (or any other string of characters).

<u>NAME OF FORM</u>	<u>VALUE</u>
M	4
SUM	6
2	10
FIVE	9

Suppose you wrote the following code:

#(PS: #(AN:M:3) #(AD:2:5) #(AD: #(2):3) #(AN:FIVE:6))

The printed result would be 771315

Forms storage would look like this:

NAME OF FORM	VALUE
M	7
SUM	6
2	10
FIVE	15

For practice, try using the new values in forms storage. Remember that Define String (DS) has no value.

#(PS:#(DS:#(M):#(AD:#(2):6))#(AN:2:2)#(DS:12:5)#(AN:#(2):3))

Printed result: 128

Forms storage would now look like this:

NAME OF FORM	VALUE
M	7
SUM	6
2	12
FIVE	15
7	16
12	8

Let's trace that out in case you are having trouble.

#(PS:#(DS:7:#(AD:10:6))#(AN:2:2)#(DS:12:5)#(AN:#(2):3)) evaluate #(7) & #(2)
 #(PS:12#(DS:12:5)#(AN:#(2):3)) evaluate (AD), do (DS) do (AN)
 #(PS:12#(AN:12:3)) do (DS) and evaluate #(2)
 #(PS:128) do (AN)

The other arithmetic functions are analogous:

arithmetic done	function producing changes in forms Has a name argument	function producing no changes in forms Both arguments are numbers
ADD SUBTRACT MULTIPLY DIVIDE	#(AN:N:D) #(SN:N:D) #(MN:N:D) #(DN:N:D)	#(AD:D1:D2) #(SU:D1:D2) #(ML:D1:D2) #(DV:D1:D2)

In subtractions, (SN) D is subtracted from the contents of N
 (SU) D2 is subtracted from D1

In divisions, (DN) the contents of N is divided by D
 (DV) D1 is divided by D2

Only whole numbers are allowed. The result is the largest whole number in the quotient.

<u>FUNCTION</u>	<u>REPLACED BY</u>
#(DV:6:2)	3
#(DV:9:2)	4
#(DV:100:91)	1
#(DV:5:-2)	-2
#(DV:-4:-3)	1

Start with the last forms storage, and find the results of

#(PSL#(SN:#(M):#(ML:#(7):2))#(DN:FIVE:#(7)))

printed: -250

New forms storage:

<u>NAME OF FORM</u>	<u>VALUE</u>
M	7
SUM	6
2	12
FIVE	0
7	-25
12	8

Tracing in detail:

#(PS:#(SN:7:#(ML:16:2))#(DN:FIVE:#(7)))

#(PS:#(SN:7:32)#(DN:FIVE:#(7)))

#(PS:-25#(DN:FIVE:-25))

#(PS:-250)

On all eight arithmetic functions, there can be a fourth argument, Z. #(AD:D1:D2:Z). The function is replaced by Z if the arithmetic result causes overflow. The maximum value will depend on the particular version of GLURP. At the ISVD, overflow was any number larger than 13 digits. Each individual argument must be 7 or fewer digits. Z can be any string including a function or the special form *stepno. Usually, a function is protected by parentheses, as in

#(PS:THE SUM IS #(AD:#(N):#(MONEY):(#(DS:FLAG:OVERFLOW)INCREDIBLE)))

Section B2 Form Storage Manipulation

Intermediate

a. Forms Pointer

There are a large number of functions that manipulate forms. Forms pointers and segment gaps must both be clear before you can understand the functions for forms manipulation available in GLURP. Every form has associated with it a pointer. When a form is defined, the pointer points to the first character of the value string. The pointer c can be moved by GLURP functions. We represent the pointer by underlining the character to which it points.

<u>FORM NAME</u>	<u>VALUE</u>	
FAUCET	<u>DRIPPY</u>	pointer points to D as when initially defined
HALITOSIS	<u>ALONE</u>	pointer has been moved to point to O

Let us consider a function that moves the forms pointer.

CALL CHARACTER #(CC:N:Z)

N is a name as usual. The value of the function is the character under the forms pointer of N. The pointer is advanced one character. If there is no character under the pointer (it is past the end), the value is Z and there is no change to forms storage. In the example of forms storage given above, the function

#(:#(CC:FAUCET)#(CC:HALITOSIS:NOW)#(CC:HALITOSIS: NOW)#(CC:HALITOSIS: NOW)
#(CC:HALITOSIS: NOW))

would print: DONE NOW

and change forms storage as follows:

<u>NAME OF FORM</u>	<u>VALUE</u>	
FAUCET	<u>DRIPPY</u>	
HALITOSIS	<u>ALONE_</u>	pointer is now beyond the last character

b. Segment Gaps

Intermediate

Every form has a pointer.. Forms can have one or more segment gaps. You can create segment gaps and fill them in using GLURP functions. Segment gaps are created using

SEGMENT STRING #(SS:N:X1:X2:X3:....)

N is the name of a form. Each X is a string of characters. Rather than

formally list the properties of Segment String (SS), we will develop them in a series of examples.

<u>NAME OF FORM</u>	<u>VALUE</u>
QUES	WHO SAYS SO

The form above could be segmented using

#(SS:QUES:A)

Given that function, GLURP would search the form named QUES for the letter A. When it found it, it would replace that letter by a type 1 segment gap. We symbolize a type 1 segment gap by @1. The resulting form would look like this:

<u>NAME OF FORM</u>	<u>VALUE</u>
QUES	WHO S@1YS SO

The form is changed. You can now ask for anything to be inserted into the segment gap.

#(PS: #(QUES: ANYTHING))

prints out: WHO SANYTHINGYS SO

The form is not changed by this printing operation. Notice that things to go into a type 1 segment gap are written after the name and a colon. Since the form is not changed, you can print it out another way.

#(PS: #(QUES: ANOTHER WAY))

prints out: WHO SANOTHER WAYYS SO

To get a bit fancier and to illustrate the earlier point that function can be written anywhere a string can be written, try this:

#(PS: #(QUES: #(QUES: WOW)))

prints out: WHO SWHOSWOWYS SOYS SO

To add some variety and practice try this:

#(DS: BAD: BANANA) #(SS: BAD: N)

<u>NAME OF FORM</u>	<u>VALUE</u>
QUES	WHO S@1YS SO
BAD	BA@1A@1A

Notice that both letters N are replaced by a type 1 segment gap.

#(PS: #(BAD: MAN))

prints out: BAMANAMANA

#(PS: #(BAD: #(QUES: #(BAD: WOW))))

prints out: BAWHO SBAWOWAWOWAYS SOAWHO SBAWOWAWOWAYS SOA

So far we have used only one segment gap (a type 1). You can create different types of gaps with one segment string function. Other types are symbolized by @1, @2, @3, etc. for types 1, 2, 3, etc.

#(DS:CAT:SIAMESE)#(SS:CAT:S:E:M)

NAME OF FORM	VALUE
CAT	@1IA@3@2@1@2
QUES	WHO S@1YS SO
BAD	BA@1A@1A

The different types of segment gaps correspond to the order in which the arguments of the Segment String (SS) function are written. In this case, @1 corresponds to S, @2 to E, and @3 to M. The gaps are filled in in the order of their types.

#(PS: #(CAT:THIS:IS:NEAT))

The letters THIS will replace each @1, IS replace each @2 and NEAT replace each @3.

prints out: THISIANEATISTHISIS

Combining everything

#(PS: #(CAT: #(BAD:MAR): #(BAD:L):N))

prints out: BAMARAMARAIANBALALABAMARMARABALALA

An intermediate step in the evaluation reads:

#(PS: #(CAT:BAMARAMARA: BALALA: N))

One last point about Segment String (SS). Each occurrence of string X1 is removed before the form is searched for X2. The processing goes from left to right. As an illustration of this rule

#(DS:FRUIT: BANANA) #(SS:FRUIT:ANA:A) #(DS:OOPS: BANANA) #(SS:OOPS:A:ANA)

NAME OF FORM	VALUE
FRUIT	B@1N@2
OOPS	B@1N@1N@1

Let us summarize the creation of segments and their use.

- 1.) Forms have no segment gaps when defined.
- 2.) Segment gaps are created using Segment String (SS).
 - A) Any number of gaps can be created at once.
 - B) Any number of characters can be segmented out.
 - C) Each occurrence of string X1 is removed before String X2 is used.
 - D) Processing is from left to right.

3.) Once a string is segmented, it is called using

#(N:X1:X2)

where there are as many X's as there were in the original segmenting function. The X's may have no characters, in which case two consecutive colons are used.

4.) Calling a form has no effect on its segment gaps.

c. Forms Pointer Functions

Intermediate

We already discussed one function that changes the forms pointer, Call Character (CC). There are several others.

CALL N CHARACTERS

#(CN:N:D:Z)

The function is replaced by the next D characters of the form N. The pointer is advanced by D characters. Negative values of D are interpreted as the D characters before the pointer. If D characters are not available, the value is Z and forms storage is unchanged. Call N Characters (CN) is fairly complicated. The examples should help clarify it.

<u>NAME OF FORM</u>	<u>VALUE</u>
MMM	ABCDEF

<u>FUNCTION</u>	<u>VALUE</u>	<u>NEW FORM VALUE</u>
#(CN:MMM:4)	ABCD	ABCDEF
#(CN:MMM:-2:BAD)	CD	ABCDEF
#(CN:MMM:-2:BAD)	AB	ABCDEF
#(CN:MMM:6:BAD)	ABCDEF	ABCDEF
#(CN:MMM:2:BAD)	BAD	ABCDEF

INITIAL

#(IN:N:X:Z)

The form N is searched for substring X beginning at the forms pointer. If X is found, the value is that part of N from the forms pointer up to the first character of X in the match. The pointer is moved to the character beyond the match. If no match is found, the value is Z and the forms pointer is unchanged.

<u>NAME OF FORM</u>	<u>VALUE</u>
MMM	ABCDEF

<u>FUNCTION</u>	<u>VALUE</u>	<u>NEW FORM VALUE</u>
#(IN:MMM:B:NO)	A	ABCDEFGG
#(IN:MMM:C:NO)		ABCDEFGG no characters in value
#(IN:MMM:AB:NO)	NO	ABCDEFGG
#(IN:MMM:FG:NO)	DE	ABCDEFGG_ pointer now beyond the end

CALL RESTORE #(CR:N)

This function restores the forms pointer of form N to the first character of the string. It has no value.

CALL SEGMENT #(CS:N:Z)

The value of the function is the substring of N from the current position of the forms pointer to the next segment gap. The pointer is moved to the character beyond the segment gap. If no characters remain, the value is Z. If no segment gap remains in the form, the value is the rest of the form.

<u>NAME OF FORM</u>	<u>VALUE</u>
BLAH	HI@1 DI@2@2DLEY B@3P WOW

<u>FUNCTION</u>	<u>VALUE</u>	<u>NEW FORM VALUE</u>
#(CS:BLAH:BAD)	HI	HI@1 DI@2@2DLEY B@3P WOW
#(CS:BLAH:BAD)	ADI	HI@1 DI@2@2DLEY B@3P WOW value includes space
#(CS:BLAH:BAD)		HI@1 DI@2@2DLEY B@3P WOW no characters in value
#(CS:BLAH:BAD)	DLEY B	HI@1 DI@2@2DLEY B@3P WOW
#(CS:BLAH:BAD)	P WOW	HI@1 DI@2@2DLEY B@3P WOW_
#(CS:BLAH:BAD)	BAD	HI@1 DI@2@2DLEY B@3P WOW_

Section B3 List Manipulation

Intermediate

a. Introduction

Three list functions change the forms pointers also. When using the list functions, you consider the value of a form as a series of items separated by one or more spaces. Several examples will clarify what this this means.

<u>NAME OF FORM</u>	<u>VALUE</u>
LINK	ORIENTIA050.00ΔΔSTOPΔΔΔ020.00ΔΔΔOMINORA100.00
STORELIST	CHEESE MILK EGGS CHICKEN PEAS MUSHROOMS

LINK has six items, some with more than one space inbetween. STORELIST also contains six items.

b. New Top

Intermediate

NEW TOP

#(NT:N:X)

The form named N is recreated with the string X and a space at its beginning (top). Any part of the value of N before the forms pointer is lost. X may contain spaces, and then is considered more than one item. The new forms pointer is to the first character of the new form. The function has no value.

#(NT:LINK:DATAΔΔΔ300.00) #(NT:STORELIST:BUTTER)

<u>NAME OF FORM</u>	<u>VALUE</u>
LINK	DATAΔΔΔ300.00ΔORIENTIA050.00ΔΔSTOPΔΔΔ020.00ΔΔΔOMINORA100.00
STORELIST	BUTTER EGGS CHICKEN PEAS MUSHROOMS

c. New Bottom

Intermediate

NEW BOTTOM

#(NB:N:X)

The form N is recreated with a space followed by X at the end (bottom). The forms pointer is not changed. The function has no value.

d. Pop Top

Intermediate

POP TOP

\$(PT:N:X:Z)

The function is replaced by the item following the forms pointer on list N. The forms pointer is advanced to the space following the next item. The value never includes any spaces. If no more items remain on the list, the value is Z and forms storage is unchanged.

Other functions treat forms as lists of items separated by spaces. These are complex functions, designed to treat lists of English words inside of a script.

e. Delete Item

Intermediate

DELETE ITEM

\$(DI:N:X1:X2:X3:...)

Form N is treated as a list of items. If X1 occurs as an item of the list, it is replaced by a single space. X2, X3, etc. are treated similarly. It is not enough for the letters of X1 to occur in order. They must be a complete item on the list N before they will be deleted. The function has no value and the forms pointer is unchanged.

f. Using the List Functions

Intermediate

Let's play with these new functions for a few minutes to get an idea how they work. Begin with STORELIST, as it was last defined. Suppose that a typical housewife went to the store to buy the things on STORELIST. As she walked around, she bought things, and created a new list of them:

\$(NT:BOUGHT:PEAS EGGS MILK STEAK ANCHOVIES)

What else is there to buy?

\$(PT:STORELIST:NO MORE)

value is BUTTER

NAME OF FORM

VALUE

STORELIST

BUTTER EGGS CHICKEN PEAS MUSHROOMS

BOUGHT

PEAS EGGS MILK STEAK ANCHOVIES

So she bought butter, and listed it.

\$(NB:BOUGHT:BUTTER)

What else?

\$(PT:STORELIST:NO MORE) value is EGGS

Didn't I already have that?

\$(IN:BOUGHT:EGGS:NOPE) value is PEAS

NAME OF FORM	VALUE
STORELIST	BUTTER EGGS_CHICKEN PEAS MUSHROOMS
BOUGHT	PEAS EGGS_MILK STEAK ANCHOVIES BUTTER

That method of getting the other items is not so good. Better to scratch them off the STORELIST as they are purchased. There are at least two ways to use the BOUGHT list to remove items from STORELIST.

Method One:

```

.
.
.
100.00 $(CR:BOUGHT)
110.00 $(DI:STORELIST:$(PT:BOUGHT:*120.00))$(GO:110.00)
120.00 $(CR:BOUGHT)

```

Method Two:

```

.
.
.
100.00 $(SS:BOUGHT: )$(DI:STORELIST:$(BOUGHT:(:)))
110.00 $(DS:BOUGHT:$(BOUGHT: ))

```

Method one assures you that it will work regardless of the number of spaces in BOUGHT between items. This is because Pop Top (PT) never has spaces in its value. Method two is faster and more elegant, since it does not depend on a loop back to the same step. The position of the forms pointer is lost in both methods.

Using either method, the list will look as follows:

NAME OF FORM	VALUE
STORELIST	_CHICKEN MUSHROOMS
BOUGHT	_PEAS EGGS MILK STEAK ANCHOVIES BUTTER

Section B4 List Comparisons

Intermediate

a. Introduction

Two special comparison functions treat forms as lists of items. These compare the two forms and test if the lists are the same. Order of items on the list is irrelevant; spaces after the one required for separation are also irrelevant. Examples of lists that are equal in this sense are included in forms storage given below:

<u>NAME OF FORM</u>	<u>VALUE</u>
NORMAL	THE DOG BIT A BOY
STRANGE	THE BOY BIT A DOG
SHOP	FOOD HARDWARE PIPES MONEY
STORE	HARDWARE PIPES MONEY FOOD
LIST1	BUTTER BREAD JAM EGGS SALT PEPPER
LIST2	BUTTER JAM SALT PEPPER
LIST3	BACON EGGS SALT PEPPER PICKLES

Lists NORMAL and STRANGE, and SHOP and STORE are identical for these list comparison functions. Let's look in detail at them.

b. Compare for Equalities

COMPARE FOR EQUALITIES #(CE:N1:N2:Z:N3)

Word list N1 is compared item by item to list N2. If all items on N1 are also on N2, the value of the function is Z and N3 is unchanged. If there are words on N1 not also on N2, then those items in both lists are put into a list named N3. Previous definitions of N3 are lost.

c. Compare for Inequalities

COMPARE FOR INEQUALITIES #(CI:N1:N2:Z:N3)

Similar to Compare for Equalities (CE), but N3 contains only those items on N1 but not on N2.

Using the last forms to illustrate these functions:

#(CE:LIST1:LIST2:SAME:A)	no value
#(CE:LIST1:LIST3:SAME:B)	no value
#(CE:LIST2:LIST1:SAME:C)	value is SAME
#(CI:LIST2:LIST3:SAME:D)	no value
#(CI:LIST2:LIST1:SAME:E)	value is SAME
#(CI:LIST3:LIST2:SAME:F)	no value

NAME OF FORM

VALUE

None of the original forms are changed, but several new ones are defined.

A	BUTTER JAM SALT PEPPER
B	EGGS SALT PEPPER
D	BUTTER JAM
F	EGGS BACON PICKLES

Section B5 Miscellaneous Forms Manipulation

Intermediate

a. Concatentate

CONCATENATE

#(CO:N:N1:N2:N3)

A new form named N is created by placing the contents of N1, N2, and N3 side by side in that order. Segment gaps in N1, N2, and N3 remain in the new N. The original strings are unchanged. Some of the N's can be left out.

b. Delete Definition

DELETE DEFINITION

#(DD:N1:N2:N3:....)

Forms with name N1, N2, N3, etc. are deleted from forms memory. The function has no value.

c. Selective Delete

SELECTIVE DELETE

#(SD:X)

All forms whose names begin with the substring X are deleted from memory. The function has no value.

Section B6 Printing Functions

Intermediate

You already know the characteristics of one function that prints text on the screen.

a. Print String

PRINT STRING

#(PS:X)

The string X is displayed on the CRT beginning on the next available line. In the ISVD implementation, X may not be more than 256 characters when printed. If you want the screen erased before X is printed, precede X by 'CLEAR' character $\begin{smallmatrix} C \\ L \end{smallmatrix}$. Be cautious about these, since clearing the screen may make it impossible for inquirers to read previously displayed text. Before using a CLEAR $\begin{smallmatrix} C \\ L \end{smallmatrix}$, you should be certain that a KEYBOARD has intervened since the last printed message.

b. Print Continuous

PRINT CONTINUOUS

#(PC:X)

Print Continuous has the same characteristics as Print String (PS), but the text is displayed at the next available space rather than on the next line. Print Continuous (PC) has the same relationship to Print String (PS) that CRTC has to CRT+.

c. Print List

The third print function treats forms like lists of items.

PRINT LIST

#(PL:N:X)

The form named N is printed on the CRT with the string X inserted between each item. Typically X is a carriage return, created a tabulated list format on the left margin. However, you may use any string for X.

Section B7 Branching Functions

Intermediate

a. Introduction

Several functions direct the computer to continue processing at some specific step in the same or a different script rather than go directly to the next step. In addition to these functions, other step types can change the order of execution. Any change in the normal order of sequential processing is called a branch. Let us review the branching techniques you already know.

GOTO 140.00

Branches to step 140.00 of the same script.

KEYBOARD:130.00

Branches to step 130.00 of the same script if the inquirer types @HELP. Otherwise goes to the next step.

DECOMP:rule:140.00

Branches to step 140.00 if the last inquirer response matches the 'rule'. Otherwise, it goes to the next step.

\$(EQ:A:\$(N):*120.00:*130.00)

Branches to step 120.00 if the value of N is 'A'. Otherwise, it branches to step 130.00. The asterisk branch notation can be used in any function that has a Z type argument.

Three other functions cause branching. These are all new to you, and are explained below.

b. Go

GO

\$(GO:STEPNO)

When this function is evaluated, the computer stops evaluating the current step and branches to STEPNO. The Go (GO) function can be used in some places where the asterisk notation (*STEPNO) is inappropriate.

120.00 \$(EQ:11:\$(N):\$(DD:N)\$(GO:140.00)):(\$(DS:M:NOT YET)\$(GO:130.00))

Assuming that #(N) is equal to 11, this statement goes to step 140.00 after deleting N. If #(N) is not 11, the form 'M' is defined with value 'NOT YET' and a branch to 130.00 is taken. Using the asterisk notation, you might incorrectly assume that

```
120.00 #(EQ:11:#(N):#(DD:N)*140.00):#(DS:M:NOT YET)*130.00))
```

is equivalent. However, this step would delete N if it were 11 or define M as NOT YET if #(N) is not 11. In either case, it will continue as the next step. The asterisk works only if it is the first character when it becomes the value of the function. In the step above, the two possible values are

```
#(DD:N)*140.00          and          #(DS:M:NOT YET)*130.00
```

The asterisk is not the first character. To be certain that this is clear, try an example that looks similar but behaves differently.

```
120.00 #(EQ:11:#(N):#(DD:N)*140.00:#(DS:N:1)*130.00)
```

This step will delete N and redefine it as 1. Then, if the original value of N was 11, branch to 140.00. If not, it branches to 130.00. The lack of protective parentheses strongly influences the outcome. Here the three internal functions are all evaluated before the Equals (EQ) is evaluated. The possible values for Equals (EQ) are then *140.00 and *130.00. Remember that Delete Definition (DD) and Define String (DS) has no replacement values.

c. Reset Step

RESET STEP

#(RT:STEPNO)

Reset Step (RT) will decide what step is next, but it finishes evaluating the current step before making the branch. Only the last Reset Step (RT) evaluated matters. Of course, a Go (GO) function nullifies the Reset Step (RT) by immediately branching.

```
110.00 #(RT:110.00)#(EQ:#(AN:N:1):11:#(RT:150.00)):#(DS:#(N):YES)))
```

```
#(PS:N = #(N) NOW)
```

For initial values of N less than 11, this step prints N ≠ 1 and all values up to 11, defines strings named N ≠ 1 up to 10 with values 'YES' and then branches to step 150.00. The important thing to notice is that the machine

remembers the branch to 110.00 while it does other stuff. When it encounters the instruction #(RT:150.00) for #(N) equal to 11, it forgets step 110.00 and remembers 150.00. Steps are often easier to write using Reset Step. The analog using Go (GO) is typically longer.

```
110.00 #(EQ: #(AN:N:1):11: (#(PS:N = 11 NOW) #(GO:150.00)):
      (#(DS: #(N):YES) #(PS:M = #(N) NOW) #(GO:110.00)))
```

With more complex login within the statement, the savings is even larger.

d. Execute

EXECUTE

#(XQ:X:STEPNO)

X is a scriptname of one to thirty characters (in the ISVD -- check this for your implementation). X may not include spaces, colons or parentheses. You should be certain that there is a script with the exact name X before using this function. When Execute (XQ) is evaluated, the next step is taken from script X, step STEPNO. If STEPNO is omitted, #(XQ:X), the first step of script X is taken. If X cannot be found or STEPNO does not exist in X, control passes to the STOP script, which then sends the inquirer to the script on top of the LINK list. This mechanism makes it possible to recover from errors in spelling X.

All scripts end with an Execute (XQ) function in the form #(XQ: #(PT:LINK): #(PT:LINK)). You can now understand that the function sends control to the scriptname and step number on top of the list called LINK.

a. Introduction

One intermediate level function allows you to change the contents of the KEYBOARD buffer internally and do DECOMP statements on whatever string you want.

(LK:N)

and later restore it to the KEYBOARD buffer for further processing using
#(LK:ANS).

86

Often you want to use a word that an inquirer has supplied by typing. The word might be selected by Keyboard Segment (KS). Unfortunately, you can not tell off hand if

is more appropriate. Fortunately we have just the function for it.

(AORAN:X)

The function is replaced by 'AN X' if the first letter of X is A, E, I, O, or U. Otherwise the replacement is 'A X'. Typically you would use it in a sentence like

```
#(PS:SO YOU WANT TO WORK AS #(AORAN):#(KS:2)))
```

To determine if an inquirer has completed a script which is a prerequisite to yours, you will want to use the function

```
#(TAKEN:D:STEPNO1:STEPNO2)
```

In the ISVD, each script has a number as well as a name. D is the number of the specific script in question. STEPNO1 is the next step if the script has been taken. Otherwise STEPNO2 is the next step. If STEPNO2 is omitted `$(TAKEN:D:STEPNO1:)` and the script D was not taken, control proceeds to the next sequential step. Other implementations may not use the same conventions for TAKEN.

Singulars and plurals are sometimes a problem when using words the inquirer types. To be sure of agreement, you can use the utility function

#(ENDINS:X1:X2:X3)

If the last character of the string X1 is the letter S, the function is replaced by X2. If X1 does not end in S, the value is X3. A typical use would be

#(PS:YOU ASKED FOR THE SALARY THAT #(KS:2) #(ENDINS:#(KS:2):MAKE:MAKES))

Presumably a job name either singular or plural has been selected by #(KS:2). You could get fancier combining ENDINS and AORAN to be certain of what you are saying. Notice that correct usage dictates

A CARPENTER MAKES
AN ARC WELDER MAKES
CARPENTERS MAKE

in the sentence above. To code this, still assuming #(KS:2) is a job name, you could write:

```
#(PS:YOU ASKED FOR THE SALARY THAT
  #(ENDINS:#(KS:2):(#(KS:2) MAKE):(#(AORAN:#(KS:2)) MAKES))))
```

Also correct, but less compact would be

```
#(PS:YOU ASKED FOR THE SALARY THAT
  #(ENDINS:#(KS:2):(#(KS:2)):(#(AORAN:#(KS:2))))
  #(ENDINS:#(KS:2):MAKE:MAKES))
```

The ENDINS function can also be used on verbs to assure agreement.

Sometimes you need to know how many characters a specific form has in it. This could be a measure of the verbosity of the inquirer.

COUNT #(COUNT:N)

The value of the function is the number of characters in the form N.

Certain actions require repetition. If they are expressed as functions, you can use

LOOP #(LOOP:N:D:X)

N is a form name and is used as the counter of times X has been done. D is the total number of times X is to be done. Previous values of N are lost. Consider several examples. To print 'NO NO NO NC NO NO' you could write

```
#(PS:NO NO NO NO NO NO)
but you could also write
#(LOOP:NOCOUNT:6:(#(PC:NO)))
```

Suppose you wanted to know how many commas there were in a 14 characters string named AARDVARK

```
#(DD:COMMA)#(LOOP:CNT:14:(#(EQ:#(CC:AARDVARK):,:(#(AN:COMMA:1)))))
```


To make a list of all the squares of numbers from 1 to 20, you could write the expression

```
#(DD:SQR)#(LOOP:N:20:(#(NB:SQR:#(ML:#(N):#(N))))))
```

Notice that the counter (N) is used in the string X as part of the instruction. Each of the twenty times X is done, N will have a different value.

You could do a similar calculation for the numbers 51 to 60. This time let's add up the squares.

```
#(DD:SUM)#(LOOP:N:10:(#(AN:SUM:#(ML:#(AD:50:#(N)):#(AD:50:#(N))))))
```

One last example. Suppose that NUM is a string containing a number. To computer NUM! (NUM factorial -- the product of all positive integers less than or equal to NUM), you could write

```
#(DS:FACT:1)#(LOOP:N:#(NUM):#(#MN:FACT:#(N):#(#PS:TOO BIG)#(GO:900.00))))
```

The intermediate section ends here, but some intermediates will find use for some of the advanced functions described in the next section. Most likely to be of use are the Store Record (SR) and the Fetch Record (FR). You may use those functions without learning all of the advanced material. They will be useful if you want to save some of the inquirers' answers from one session to the next.

You might want to let an inquirer see another script to explain a concept more fully. For instance, if your script discussed Legal Requirements for Jobs, an inquirer might want to see a more general script on Requirements. Use the following code to achieve this (steps 300.00 and 310.00).

FIGURE 22

SCRIPT ILLUSTRATING LINKAGE

```
.  
. .  
100.00 CRT DO YOU WANT TO SEE A SCRIPT DEFINING WHAT WE MEAN BY  
      REQUIREMENTS?  
110.00 KEYBOARD  
120.00 DECOMP:(POS):300.00  
130.00 CRT O.K. WE WILL GO ON THEN TO SPECIFIC REQUIREMENTS.  
140.00 GOTO 600.00  
150.00 CRT YOU HAVE JUST SEEN THE GENERAL DEFINITION OF REQUIREMENTS.  
      NOW LET'S CONTINUE WITH SPECIFIC LEGAL REQUIREMENTS FOR JOBS.  
160.00 GOTO 600.00  
300.00 #(NT:LINK:LEGALREQUIREMENTSFORJOBS 150.00)  
310.00 #(XQ:REQUIREMENTS)
```

After seeing REQUIREMENTS, the inquirer will see step 150.00 of LEGALREQUIREMENTSFORJOBS. You must be certain to use the correct name for your own and the other script. To be absolutely certain that the inquirer will return to your script, you can also include

```
305.00 #(NT:STOP:LEGALREQUIREMENTSFORJOBS 150.00)
```

INTERMEDIATES MAY STOP HERE.

Section C Advanced Weird

Section C1 Neutral Functions

Advanced

Several new concepts are necessary for advanced GLURP. These include fetching and storing on disc memory, decoding, manipulating bit strings and using neutral functions. Let us first discuss the notational extension called neutral functions to show how useful it can be in scripts. To motivate the details below, consider the following example.

<u>NAME OF FORM</u>	<u>VALUE</u>
A	#(B)
B	#(C)
C	#(AB)
AB	A

Suppose you wanted to see the contents of form B. You could try several things you already know.

<u>FUNCTION</u>	<u>PRINTS</u>
#(PS:#(B))	A
#(PS:B)	B
#(PS:(#(B)))	#(B)

So far, you have no way to see the contents of B. But, of course, there is a way or we wouldn't be going through all of this.

#(PS:##(B)) #(C)

You can read the ##(notation as 'get the value, but do not process it further.' Consider other examples:

#(DS:AC:##(A)##(C))

<u>NAME OF FORM</u>	<u>VALUE</u>
AC	#(B)#(AB)

#(DS:AA:#(A)##(A)##(A))

<u>NAME OF FORM</u>	<u>VALUE</u>
AA	A#(B)#(A)

#(DS:#(A)#(B)##(C):#(A)##(B)#(A)))

<u>NAME OF FORM</u>	<u>VALUE</u>
AA#(AB)	AA#(B)#(A)

#(DS:#(A)##(A)##(C)):##(AA)#(AA)##(AA)))

<u>NAME OF FORM</u>	<u>VALUE</u>
AAAA	A#(B)#(A)AAA#(AA)

$$\#(DS:A\#\#(A):(\#(AB)))$$

A

Looking at the active pointer ('A'), notice that there is an active function #(. Move the pointers

$$DS : A \# \# (A) : (\# (AB))$$

A

'FA

D

$$S : A \# \# (A) : (\# (AB))$$

'N

A

'FA

DS

$$:A\#\#(A):(\#(AB)))$$

'N

'A

'FA'A

DS

$$A\#\#(A):(\#(AB)))$$

'N

'A

'FA'A

DS A

$$\#\#(A)(\#(AB)))$$

'N

A

'FA'A'FN

DS A

$$A) : (\#(AB))$$

1000

'A

Move a non-syntactic character.

'FA'A'FN
 DS A A):(#(AB)))
 'N 'A

Notice the) at 'A. End the last function. Do the last function. Since it was neutral, (FN), put the result on the neutral buffer.

'FA'A
 DS A#(B) :(#(AB)))
 'N 'A

Mark an argument.

'FA'A 'A
 DS A#(B) (#(AB)))
 'N 'A

The parenthesis) at 'A is a non-function parenthesis, meaning protect. Change to protect mode, and move characters to find the balancing right parenthesis.

PAREN COUNT = 1 PROTECT = ON
 'FA'A 'A
 DS A#(B) #(AB)))
 'N 'A

Move the #.

PAREN COUNT = 1 PROTECT = ON
 'FA'A 'A
 DS A#(B) # (AB)))
 'N 'A

This left parenthesis is not the balancing right parenthesis we want.

PAREN COUNT = 2 PROTECT = ON
 'FA'A 'A
 DS A#(B) #(AB)))
 'N 'A

Move the next two non-parentheses.

PAREN COUNT = 2 PROTECT = ON
 'FA'A 'A
 DS A#(B) #(AB)))
 'N 'A

This right parenthesis reduces the count, but not to zero.

PAREN COUNT = 1 PROTECT = ON
 'FA'A 'A
 DS A#(B) #(AB)))
 'N 'A

The right parenthesis here is the one which reduces the count to zero. Turn off the protect mode, but do not move this parenthesis.

PAREN COUNT = 0

PROTECT = OFF

'FA'A	'A
DS A#(B) #(AB)	)
'N	'A

The final parenthesis (in PROTECT = OFF mode) means end the last function. The name of the function is from the function marker to the first argument -- DS. The first argument is A#(B). The last argument is #(AB).

NAME OF FORM	VALUE
A#(B)	#(AB)

Messy, huh? A little faster this time. Start with this

	#(EQ:#(C):##(#(B)):(WOW):(##(C)))
'N	'A

Mark an active function and move two non-syntactic characters over.

'FA	
EQ	:#(C):##(#(B)):(WOW):(##(C)))
'N	'A

Mark an argument and a new function.

'FA'A'FA	
EQ	C):##(#(B)):(WOW):(##(C)))
'N	'A

Move the non-syntactic character, then do the function. The function is active (FA), so place the result on the top of the active buffer.

'FA'A	
EQ	#(AB):##(#(B)):(WOW):(##(C)))
'N	'A

Another active function and a few other characters.

'FA'A'FA	
EQ AB	):##(#(B)):(WOW):(##(C)))
'N	'A

Evaluate the result of an active function and put the result on the active buffer.

'FA'A	
EQ	A:##(#(B)):(WOW):(##(C)))
'N	'A

Mark a new argument after moving one character.

'FA'A'A	
EQ A	##(#(B)):(WOW):(##(C)))
'N	'A

Two functions in a row, one neutral, the other active.

'FA'A'A'FN'FA

EQ A	B)):(WOW):(##(C)))
'N	'A

Move B and evaluate a function actively.

'FA'A'A'FN

EQ A	#(C)):(WOW):(##(C)))
'N	'A

Go through the same stuff for #(C).

'FA'A'A'FN

EQ A	#(AB)):(WOW):(##(C)))
'N	'A

Once more.

'FA'A'A'FN

EQ A #(B)	A):(WOW):(##(C)))
'N	'A

Move the character (A) and do the next function neutrally.

'FA'A'A

EQ A #(B)	:(WOW):(##(C)))
'N	'A

Mark a new argument, then protect, move three characters (WOW) and then un-protect.

'FA'A'A 'A

EQ A #(B) WOW	:(##(C)))
'N	'A

New argument, then handle the protect mode.

'FA'A'A 'A 'A

EQ A #(B) WOW ##(C)	)
'N	'A

Do the function. It is an EQ function. The two arguments are not equal, so the value (on the active buffer) is the last argument.

	##(C)
'N	'A

Do the neutral function and leave the result on the neutral buffer.

#(AB)	
'N	'A

Section C2 The TRAC Algorithm

Advanced

Appendix F contains a formal description of the original TRAC algorithm. Please note that rule 3 of the algorithm does not apply in the GLURP system and that in rule 4 we use a colon, not a comma.

Section C3 Use of Neutral Functions

Advanced

Some items in the data base may contain embedded parentheses or even unbalanced parentheses. The neutral string evaluation allows you to handle these problems. Consider the following examples.

Example One:

Some possible ATTRIBUTE values contain parentheses. You want to print the values of the ATTRIBUTE for different RECORDS. The RECORD name is contained in a string called CURRENT RECORD, obtained from the inquirer. If you write

```
$(PS:$(FR:MILITARY:$(CURRENT RECORD):DESCRIPTION))
```

the balanced parentheses in the ATTRIBUTE DESCRIPTION will be lost in the printing and the unbalanced parentheses will cause an error message. To handle this case properly, you must avoid having the value of the Fetch Record (FR) re-evaluated by writing

```
$(PS:##(FR:MILITARY:$(CURRENT RECORD):DESCRIPTION))
```

To be perfectly safe even from the inquirer typing in RECORD names with parentheses, you could write the script as follows:

FIGURE 23

A SCRIPT REQUIRING NEUTRAL STRINGS

```
.  
. .  
100.00 CRT PLEASE TYPE THE NAME OF A MILITARY CAREER.  
110.00 KEYBOARD  
115.00 $(DS:CURRENT RECORD:##(CK))  
116.00 $(PS:THE DESCRIPTION OF A ##(CURRENT RECORD) IS  
      ##(FR:MILITARY:##(CURRENT RECORD):DESCRIPTION:*130.00))  
120.00 CRT DO YOU WANT ANY OF THESE?  
121.00 KEYBOARD  
122.00 DECOMP:(POS),(NEG),Ø:100.00:150.00:115.00  
130.00 CRT SORRY, THAT IS NOT A MILITARY RECORD NAME.  
140.00 GOTO 120.00  
. . .
```

Example Two:

A second example involves manipulation of forms storage. Forms can themselves be functions, and this can lead to difficulty unless one is careful. Consider the following segment from the EDIT script as an example of the use of neutral strings.

FIGURE 24

A SEGMENT OF EDIT

[Note: OLD contains the contents of some arbitrary script step, and is to be changed by segmenting out old parts and substituting new parts. The following script segment does the segmenting on up to 6 substrings with normal text or one containing unbalanced parentheses.

```

.
.
.
      C
350.00  #(PS:L##(OLD))
351.00  CRT+  TYPE IN THE SUBSTRINGS ON WHICH YOU WANT TO SEGMENT.
          SEPARATE BY COLONS--UP TO 6.  USE "TRAC " FOLLOWED
          BY SUBSTRING IF IT CONTAINS A LITERAL: ( ).
352.00  KEYBOARD
353.00  DECOMP:TRAC 0,0:354.00:360.00
354.00  #(DS:SEG:##(OLD))
          #(DS:NEW:##(CK))
          #(IN:NEW: :(#(PS:NO SPACE. TRY AGAIN)#(GO:352.00)))
          #(SS:SEG:##(CS:NEW))#(GO:361.00)
360.00  #(DS:SEG:##(OLD))#(SS:SEG:#(CK))
      C
361.00  #(PS:L##(SEG):*1*:*2*:*3*:*4*:*5*:*6*)
365.00  CRT+  YOU MAY NOW TYPE IN YOUR SUBSTITUTIONS.  USE "TRAC "
          FOR LITERAL : ( ).
370.00  KEYBOARD:350.00
375.00  DECOMP:TRAC 0,0:376.50:380.00
376.50  #(DS:NEW:##(CK))
          #(IN:NEW: :(#(PS:NO SPACE. TRY AGAIN)#(GO:370.00)))
          #(DS:NEW:##(SEG:##(CS:NEW)))#(GO:400.00)
390.00  #(DS:NEW:##(SEG:#(CK)))
      C
400.00  #(PS:L##(NEW))
405.00  CRT+  IS THAT CORRECT?
          A YES, STORE IT.
          B NO, I'LL TYPE SUBSTITUTIONS AGAIN.
          C NO, I'LL TYPE SEGMENTS AGAIN.
410.00  KEYBOARD
.
.
.

```

A typical protocol from the EDIT script follows:

MACHINE: CRT THERE IS A SPELLING MISTAKE HEAR.
TYPE IN THE SUBSTRINGS ON WHICH YOU WANT TO SEGMENT. SEPARATE BY COLONS--UP TO 6.
USE "TRAC " FOLLOWED BY SUBSTRING IF IT CONTAINS A LITERAL : ().
INQUIRER: NI:EAR
MACHINE: CRT THERE IS A SPELLING MISTAKE HERE.
IS THAT CORRECT?
A YES, STORE IT
B NO, I'LL TYPE SUBSTITUTIONS AGAIN
C NO, I'LL TYPE SEGMENTS AGAIN
INQUIRER: A

A second protocol, this time using the "TRAC " feature follows:

MACHINE: #(EQ: #(YES): #(NOPE)):TURE:FALSE)
TYPE IN THE SUBSTRINGS ON WHICH YOU WANT TO SEGMENT. SEPARATE BY COLONS--UP TO 6.
USE "TRAC " FOLLOWED BY SUBSTRING IF IT CONTAINS A LITERAL : ()?
INQUIRER: TRAC):TUR
MACHINE: #(EQ: #(YES): #(NOPE)*1:E:FALSE)
YOU MAY NOW TYPE IN YOUR SUBSTITUTIONS
USE "TRAC " FOR LITERAL : ().
INQUIRER: (:TRU)
MACHINE: #(EQ: #(YES): #(NOPE):TRUE:FALSE)

.
.
.

Notice that the substitution did not use the "TRAC " feature, but used protective parentheses instead.

Section C4 Binary Strings

Certain forms in memory will be binary strings. These are sets of ones and zeros such as:

NAME OF FORM	VALUE
A	10101010100001010100110101011101011111
B	111110001011101111110110010111011110101
C	110
D	0

No specific requirements are made on how binary strings are used. In the ISVD, it was convenient to use binary strings to indicate if certain conditions had occurred or if items of data were members of defined sets.

For example, the binary string

ANS 10110111011000001

might mean that questions 1, 3, 4, 6, 8, 10, 11, and 15 were all answered correctly. This corresponds to bits 1, 3, 4, 6, 8, 10, 11, and 16 being ones. Each position in a binary string is called a bit. Notice that counting bits in binary strings proceeds from left to right. The binary string

OUTDOORS 1100010110001

might mean that jobs 1, 3, 6, 8, 9, 11, 12, and 16 involved outdoor work.

The binary string

STAKEN 10001000100101

might mean that the inquirer has taken the scripts numbered 1, 5, 9, 12, and 14.

In the ISVD implementation of GLURP (though not necessarily others) binary strings were not equivalent to regular strings. You could not write the function

#(DS:A:1001011110)

and expect the form A to behave as a binary string. Binary strings were obtained by reference to the data base or by combining existing strings using functions like Binary And (BA), Binary Or (BO), or Binary Complement (BC). Binary strings can also be created by Turning On (TO) the desired bits. In general, binary strings can not be printed, and are used internally only. (Check your particular implementation for this feature.)

To create binary strings in memory, you can use

TURN ON

#(TO:N:D)

If the form N is defined, bit D is forced to value 1. Other bits are unchanged. If N is undefined, it is created with a single bit on. The function has no replacement value.

The opposite function creates 0's in a form.

TURN OFF

\$(TF:N:D)

If the form N is undefined, there is no action. If N is defined, bit D is forced to have value 0. N is never made longer to accomplish this since all bits beyond the end of N are presumed to have value 0.

You will often want to determine if a specific bit is 1 or 0.

TEST BIT

\$(TB:N:D:Z1:Z2)

Bit D in form N is examined. If it is a 1, the value of the function is Z1. If it is a 0, the value of the function is Z2. Forms storage is unchanged.

The three binary functions already described all an author to keep track of an inquirer's progress through complex script material. By assigning a bit number in a string (e.g. RECOMPS) to each CRT-KEYBOARD pair in a script, the script can tell if an inquirer has seen that text before. If so, an appropriate rephrasing can be presented. The following script segment illustrates one method for accomplishing this.

FIGURE 25

SCRIPT CHANGING WITH INQUIRER EXPERIENCE

```
.
.
.
100.00 $(TB:RECOMPS:13:*150.00)
110.00 CRT SOME JOBS REQUIRE INDOOR WORK: OTHERS OUTDOOR. WHICH DO YOU
      PREFER?
111.00 $(TO:RECOMPS:13)
120.00 KEYBOARD
130.00 DECOMP:0 IN$ 0,0 OUT$ 0:199.00:200.00
135.00 CRT DO YOU PREFER INDOOR OR OUTDOOR?
137.00 GOTO 120.00
150.00 CRT WE ASKED BEFORE ABOUT JOBS INDOORS. DO YOU WANT TO WORK
      OUTSIDE OR INSIDE?
160.00 $(TF:RECOMPS:13)
165.00 GOTO 120.00
.
.
.
```

In the example above, you must imagine that the inquirer will use the script several times, and particularly encounter step 100.00 more than once. On the second time, he will get the statement at step 150.00 instead of that in 110.00. From then on, the statements at 110.00 and 150.00 will alternate at each presentation.

Other binary functions allow you to manipulate and combine binary strings. Suppose that forms storage includes the following binary strings.

<u>NAME OF FORM</u>	<u>VALUE</u>
A	11011011
B	10101010
C	01011011
D	11011011
E	10111000
F	1101

To count the number of ones in a binary string, use the function

COUNT BINARY

#(CB:N)

The function is replaced by the decimal integer equal to the number of ones in the form N. Forms storage is unchanged. Using the forms above for examples, we have the following:

<u>FUNCTION</u>	<u>VALUE</u>
#(CB:A)	6
#(CB:B)	4
#(CB:C)	5

Sometimes you want to change each bit in the form. This makes those that were ones into zeros, and those that were zeros into ones. If a string represents the correct answers an inquirer gave, changing each bit will represent the wrong answers. This process is called complementing, and is accomplished with the function

BINARY COMPLEMENT

#(BC:N)

The value of each bit is reversed. The function has no value. The form N is changed to the complemented version.

After the following line of code, forms storage would be changed to the new version below.

#(BC:A) #(BC:E)

<u>NAME OF FORM</u>	<u>VALUE</u>
A	00100100
B	10101010
C	01011011
D	11011011
E	01000111
F	1101

In the ISVD implementation, this function is a problem. Each binary string is a multiple of 8 bits long, regardless of your intentions. Additional bits are on the right hand end, and are all zeros. While form F is written 1101, in the ISVD implementation, it is represented by 11010000. In complementing, this has strange consequences. For example,

#(BC:F)

leads to the new value

<u>NAME OF FORM</u>	<u>VALUE</u>
F	00101111

In other implementations, only the explicit length would be complemented.

The next three functions perform logical combinations of different binary forms. Let's see how they work, then illustrate some possible uses of the functions.

BINARY AND

#(BA:N1:N2)

Forms N1 and N2 are ANDed together bit by bit. The resulting bit string replaces N1. The new length is the same as the longer of N1 and N2. N2 is unchanged. The function has no replacement value.

Logical ANDing is accomplished by the following procedure: Go from the first to the last bit doing the process below.

If both bits in question are ones, the result is a one. Otherwise, the result is a zero. Both N1 and N2 must have a one for a one to appear in the result. The same decision process is applied to every bit in the strings.

After

#(BA:A:B)#(BA:D:E)

Forms storage would contain

<u>NAME OF FORM</u>	<u>VALUE</u>
A	00100000
B	10101010
C	01011011
D	01000011
E	01000111
F	1101

BINARY OR

#(BO:N1:N2)

Forms N1 and N2 are ORed together bit by bit. The resulting string replaces N1. The new length is the same as the longer of N1 and N2. N2 is unchanged. The function has no replacement value.

Logical ORing is accomplished by the following process.

If either or both of the bits in question are ones, the result bit is a one.

If both of the bits in question are zeros, the resulting bit is a zero.

Either N1 or N2 (or both) must have a one for a one to appear in the result.

After

#(BO:F:E)#(BO:B:C)

Forms storage would contain

NAME OF FORM	VALUE
A	00100000
B	11111011
C	01011011
D	01000011
E	01000111
F	11010111

BINARY EXCLUSIVE OR

#(BE:N1:N2)

Form N1 and N2 are EXCLUSIVE ORed together bitwise. The resulting string replaces N1. The new length of N1 is the same as the longer of N1 and N2. N2 is unchanged. Logical EXCLUSIVE ORing has the following property. If the two bits in question differ, the result is a one. If they are the same, the result is a zero. N1 or N2 but not both must have a one for the result to be a one.

Consider the functions

#(BE:A:B)#(BE:C:D)#(BE:E:F)

The new values in forms storage will be

NAME OF FORM	VALUE
A	11011011
B	11111011
C	00011000
D	01000011
E	10010000
F	11010111

Section C5 Using Binary Functions

Binary functions seem mysterious at first. How would these functions be used? What good are they for instruction? Suppose that you have the following binary strings:

NAME OF FORM	VALUE
NORTHEAST	110110111011000110100010
WEST	001001000100110000000001
LARGE	100010001000100100001000
COED	101011110111011001101011
EXPENSIVE	100100001000001001001000
CHEAP	000010000010000110000010

Each is a binary string with each bit corresponding to a school. An inquirer might specify that he wants a school either in the NORTHEAST or WEST. You could then code his request

```

#(DS:TEMP:$(NORTHEAST))$(BO:TEMP:WEST)

```

He might then further specify that the school can not be large. You code

```

#(DS:TEMP2:$(LARGE))$(BC:TEMP2)$(BA:TEMP:TEMP2)

```

He insists on girls being present. You can code

```

$(BA:TEMP:COED)

```

Finally, he wants a school neither too costly or too cheap. You code

```

$(DS:TEMP2:$(CHEAP))$(BE:TEMP2:EXPENSIVE)$(BC:TEMP2)$(BA:TEMP:TEMP2)

```

Now the critical question. Are there any schools that meet all those requirements? You respond with the code

```

$(PS:$(EQ:$(CB:TEMP):0:NO:YES))

```

How many?

```

$(PS:$(CB:TEMP))

```

Throughout each step, care was taken not to change the contents of any existing binary string. Only temporary strings TEMP and TEMP2 were changed.

Let us trace the results after each step in the above example.

```

$(DS:TEMP:$(NORTHEAST))$(BO:TEMP:WEST)

```

```

TEMP      ....  11111111111110110100011

```

```

$(DS:TEMP2:$(LARGE))$(BC:TEMP2)$(BA:TEMP:TEMP2)

```

```

TEMP2     ....  011101110111011011110111

```

```

TEMP      ....  011101110111010010100001

```

```

$(BA:TEMP:COED)

```

```

TEMP      ....  001001110111010000100011

```

```
#(DS:TEMP2:$(CHEAP))$(BE:TEMP2:EXPENSIVE)
TEMP2      ....      100110001010001111001010
#(BC:TEMP2)$(BA:TEMP:TEMP2)
TEMP2      ....      011001110101110000110101
TEMP       ....      001001110101010000100001
#(PS:$(CB:TEMP))
9  (prints)
```

Section C6 Storage -- Changing Data on the Disc

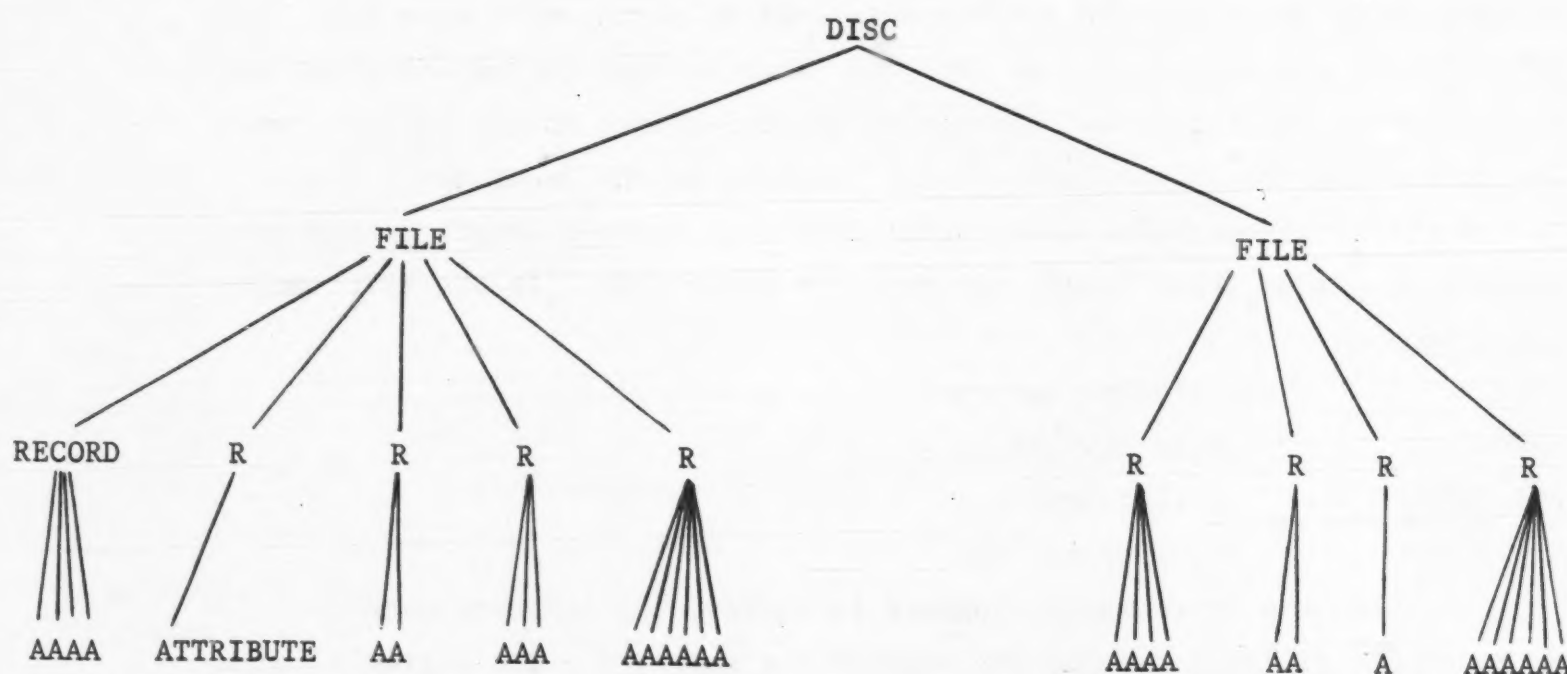
a. Introduction

Data on the disc is not the same as data in forms storage. There is more space and consequently more data on the disc. The large quantity of data available on the disc makes one name insufficient to uniquely determine the item to which you are referring. Just as people need three (or sometimes more) names to distinguish them from one another, so data on the disc need three names. These are called FILE, RECORD, and ATTRIBUTE.

The discs are filled with FILES. Each FILE has many RECORDs. Each RECORD has many ATTRIBUTES. This structure is represented in Figure 25.

FIGURE 26

DIAGRAM OF FILE STRUCTURE



To be more concrete, in the ISVD we had data about higher education including facts about colleges and trade schools. All the data together was in the Education FILE. All the data about the Fanny Farmer School of Cooking was in one RECORD as was all the data about Harvard University. The RECORDs contained ATTRIBUTES relevant to the specific kind of school. As examples:

ATTRIBUTE	WHICH RECORDs?
TUITION	Trade schools and colleges
APPLICATION FEE	Trade schools and colleges
SEX OF STUDENT BODY	colleges only
RELIGIOUS AFFILIATION	colleges only
TRADES TAUGHT	Trade schools only
LENGTH OF COURSE	Trade schools only

All college RECORDs have the same ATTRIBUTES, but differ because the ATTRIBUTES have different values.

b. Store Record

All of this discussion about data FILES on the disc introduces the function Store Record (SR), which is far more complex than any of the others.

STORE RECORD

#(SR:FILE:RECORD:ATTRIBUTE:X:Z)

This means call the string X by the compound name FILE:RECORD:ATTRIBUTE and put it on the disc. This command is used to permanently save data, usually about the inquirer. The data can later be used to individualize presentations to the inquirer. This gives the appearance of the machine remembering and can facilitate learning and interest by the inquirer.

Each of the three names still have some maximum length in any installation. Your group leader can tell you about that. In the ISVD implementation

FILE	is 1-8 characters
RECORD	1-32 characters
ATTRIBUTE	1-17 characters
X	0-255 characters

When the Store Record command is issued, the compound name FILE:RECORD:ATTRIBUTE becomes the name of the string X. The string X becomes the value of that ATTRIBUTE in that RECORD in that FILE. If for some reason

the item X cannot be stored with that name, the value of the function is Z. Normally the function has no value.

Store Record (SR) is a complex function with many conditions. It operates in the following way. If no such FILE exists, the value is Z. If no such RECORD exists in the FILE, it is created and has only the one ATTRIBUTE being stored. If the RECORD already exists, there are two possibilities:

- 1.) Within the RECORD no ATTRIBUTE already has the same name. In this case the ATTRIBUTE and its value are added to the RECORD if there is room. If there is no room, the value is Z.
- 2.) Within the RECORD, the ATTRIBUTE already exists. In this case the previous value is deleted and the new one replaces it. If the new value is longer than the old, there may not be room for it in the RECORD. In that case the value of the function is Z.

For more about the storage functions, see "Storage in the ISVD: Theory and Practice" (Roman, 1969). That paper includes details on how space is determined in a record and how to plan for efficient use of space and time.

c. Other Storage Functions

REPLACE ON THE DISC

#(RD:FILE:RECORD:ATTRIBUTE:X:Z)

Replace behaves as Store Record (SR) does if the ATTRIBUTE already exists in that RECORD in that FILE. However, if the ATTRIBUTE, RECORD and FILE are not all there, the value of the function is Z and the disc is unchanged.

INSERT ON THE DISC

#(ID:FILE:RECORD:ATTRIBUTE:X:Z)

If an item with the same full name is on the disc, the value of the function is Z and the disc is unchanged. If the FILE does not exist, the value is Z. If the RECORD does not exist, it is created. If the RECORD exists, but no such ATTRIBUTE is found, it is added to the record providing there is space. If not, the value of the function is Z and the disc is unchanged.

DELETE FROM THE DISC

#(DT:FILE:RECORD:ATTRIBUTE::Z)

If an item with that full name can not be found, the value of the function is Z and the disc is unchanged. If it is found, it is deleted

and there is no replacement value. Notice that there are two colons before the Z argument.

DELETE FROM DISC WITH FETCH #(DF:FILE:RECORD:ATTRIBUTE:N:Z)

This behaves just as Delete (DT) does except that when an item is deleted, its value becomes the value of form N. If the item is not deleted, forms storage is unchanged. If N is omitted, and the item is deleted, the value replaces the entire function and forms is unchanged.

NICK FILES

One set of FILES contains data about inquirers. Each inquirer has his own FILE. The name of the inquirer FILE is placed in a form called NICK at the beginning of each session. To store (SR) data in an inquirer FILE, use a function of the form

#(SR: #(NICK):RECORD:ATTRIBUTE:X:Z)

Fetching is done analogously. No matter who is using the script, the data will be stored appropriately. Other inquirers will never see personal data stored in this way.

d. Encoding Data

Data can be stored on the disc in three distinct forms. Simplest is to store exactly what you want to fetch. In this category, you could store summaries and inquirer makes, answers to multiple choice questions, or counts of various things. The problems with storage in this form are space required and the difficulty of interpretation by the machine later.

Data can also be coded before it is stored. You can set up a table of codes and meanings, and store the coded value rather than the actual words. An easy example would be

<u>CODE</u>	<u>MEANING</u>
1	Military
2	College
3	Trade School
4	Occupation

If the inquirer was concerned with colleges, you might write

#(SR: #(NICK):DECISION:1:2)

When you want to use this data, you could decode it by the same table.

Suppose you retrieved the stored number back from the disc using a fetch

(FR) command and put it into the form named 'ANS'. To interpret it, you could use the code

```
#(PS:YOU SAID
#(EQ: #(ANS):1:MILITARY)
#(EQ: #(ANS):2:COLLEGE)
#(EQ: #(ANS):3:TRADE SCHOOL)
#(EQ: #(ANS):4:OCCUPATION
CHOICE CONCERNED YOU)
```

A more compact form of coding is binary coding. In this form, you also make up a table, but instead of storing the number, you turn on a bit in a form to correspond to the inquirer's meaning. The college factor list is an example.

FIGURE 27

DECODE TABLE - COLFACT - LIST 16

CODE	
01	Geography
02	Type of College (Public/Private - Religious Affiliation)
03	Sex of Student Body (Men, Women, Coed)
04	Size of Student Body
05	Type of Programs (Liberal Arts, Preprofessional, etc.)
06	Financial Aid Available
07	Location (Urban, Rural)
08	Special Courses of Study
09	Extracurricular Activities
10	Academic Standard for Admission
11	Cost
12	Housing

Suppose the inquirer wrote "I want to be near home and it can't be too expensive. I need a scholarship anyway."

You could encode his essential concerns by writing

```
#(TO:VEC:1)#(TO:VEC:11)#(TO:VEC:6)
```

Some tables are used in many scripts, and are preserved on the disc itself. When a table achieves this status, the values are decoded

automatically when you fetch the data. You do not need to do a series of Equals or Compare functions to determine the meaning. Speak with your group leader to find out what tables are permanently on your FILES.

Section C7 Fetching Data from the Disc

Simple fetching allows you to bring data from the disc into forms memory where it can be examined. Data you have stored (SR), data other authors store, and data initially placed on the disc can all be fetched into forms memory. The simplest fetch is

FETCH RECORD #(FR:FILE:RECORD:ATTRIBUTE:N:Z)

If the item is not found, the value of the function is Z and forms storage is unchanged. If it is found, the form N is defined with the value found on the disc. If a table is available to decode the data in the FILE with that ATTRIBUTE, the English equivalent of the code is placed in N. If no table is available, the actual stored value is placed in N. N can be omitted from the function as in

#(FR:FILE:RECORD:ATTRIBUTE) or #(FR:FILE:RECORD:ATTRIBUTE::Z)

The function with omitted N is replaced by the value found on the disc or its English equivalent. If the data was stored for you, use this function. Whenever you are in doubt between Fetch Record (FR) and Fetch Only (FO), use Fetch Record.

FETCH ONLY #(FO:FILE:RECORD:ATTRIBUTE:N:Z)

This function acts just like Fetch Record (FR), except that items on the disc are not decoded into their English equivalent even if there is a table available. If you know that there is not a table available for decoding, use the Fetch Only (FO) command to save time. When in doubt, choose Fetch Record (FR).

DECODE BINARY #(DC:FILE:ATTRIBUTE:N:Z)

If N contains a binary string, either created on-line or previously fetched from the disc, this function translates it into its English equivalent. If it can not be translated or there are no more bits on, the value of the function is Z. Items returned are separated by commas. Only 256 characters at a time are decoded in the ISVD implementation. Issuing the command again causes the next 256 characters to be returned. The example below should make this clear.

Typically Decode Binary (DC) is used to find out what items are on a list such as jobs or colleges. Each bit corresponds to a particular

college or job. The bit number is listed in a table and the name of the job (or college) corresponds.

FIGURE 28

EDUCATIONAL INSTITUTIONS DATA BASE
COLLEGES FILE

LIST 1 - RECORD NAMES

CODE

001	Central Connecticut State College
002	Danbury State College
003	Hartford State Technical Institute
004	Manchester Community College
005	Northwestern Connecticut Community College
006	Norwalk Community College
007	Norwalk State Technical Institute
008	Southern Connecticut State College
009	Thames Valley State Technical Institute
010	United States Coast Guard Academy
011	University of Connecticut
012	Waterbury State Technical Institute
013	Willimantic State College
014	Albertus Magnus College
015	Annhurst College
016	Berkeley Divinity School
017	Bridgeport Engineering Institute
018	College of Notre Dame of Wilton
019	Connecticut College
020	Diocesan Sisters College
021	Fairfield University
022	Hartford College for Women
023	Hartford Seminary Foundation
024	Holy Apostles Seminary
025	Holy Family Seminary
026	Mitchell College

As an example, if N contained the binary string
10001100110001111

and the command

#(DC:EDUCATION:BINARY1:N)

were issued, the function would be replaced by

,CENTRAL CONNECTICUT STATE COLLEGE,NORTHWESTERN CONNECTICUT COMMUNITY COLLEGE,
NORWALK COMMUNITY COLLEGE,THAMES VALLEY STATE TECHNICAL INSTITUTE,UNITED
STATES COAST GUARD ACADEMY,ALBERTUS MAGNUS COLLEGE,ANNHURST COLLEGE,BERKELEY
DIVINITY SCHOOL

If the same command were issued again

#(DC:EDUCATION:BINARY1:N)

the function would be replaced by

,BRIDGEPORT ENGINEERING INSTITUTE

These results can be checked by using the table for BINARY1 given here as
Figure 27. The decoding would begin at the beginning of form N again if
the command

#(CR:N)

were issued. With the same string, you can get different results by decoding
in the OCCUPATION FILE.

#(DC:OCCUPATION:BINARY1:N)

,ARCWELDER,ELECTRICIAN,ACCOUNTANT,ARCHITECT,ILLUSTRATOR,DANCER,COLLEGE FACULTY
MEMBER,CHIROPRACTOR,AERONAUTICAL ENGINEER

To fully use the Decode Binary (DC) function, you must be familiar
with data base documentation or good friends with someone who is. Data base
documentation is discussed in "The Construction and Implementation of a
Data Base" (Yee, Roman, Little, and Archibald, 1969).

The last of the fetching functions is

DECODE

#(DE:FILE:ATTRIBUTE:N:Z)

This function decodes the value of the form N as if it were ATTRIBUTE
in FILE. If for some reason the decoding is impossible, the value of the
function is Z. Normally the function is replaced by the decoded value. The
Decode (DE) function allows you to interpret the same data in different ways
depending on the context. As an example, you may have a value that is either
1 or 2. In some places you want it to mean YES or NO, in others HAVE or
HAVE NOT, and in still others IS or IS NOT. The Decode (DE) function allows
you this flexibility if the tables are available on the disc.

Section C8 Special Functions for Debugging

The functions described in this chapter are rarely used in scripts, but provide help to an author trying to find out why his script does not work. They are important since very few good scripts work properly the first time. Some of the functions cause material to be written to the log tape, a record of the progress of the interaction. The log tape can be printed and examined at leisure after a debugging session. The use of these functions makes the log tape more meaningful when it is examined after the session.

LOG

\$(LG:X)

The string X is written on the log tape. X may not exceed 256 characters on the ISVD implementation. Usually X is a question or a comment about the progress of the script. A convenient way to improve the use of Log (LG) is to categorize your comments by the first two characters of the message. For example, when slides are shown on the ISVD, the function \$(LG:SL SLIDE HAS BEEN SHOWN)

executes. Any message on the log tape about slides is prefixed by SL. Similarly,

\$(LG:?? X)

means that X is a question about the working of the script. You can make up your own codes.

The Log (LG) function is often useful within a script to call particular attention to a strange response. When an inquirer gives an unrecognized response at certain key points in the script, the answer is logged.

\$(LG:CK \$(CK) WAS NOT RECOGNIZED)

LOG CRT

\$(LC)

This command causes all material printed on the screen for that inquirer to be logged as well. Only the single inquirer issuing the command is effected. Currently, the ISVD script network issues \$(LC) for all inquirers when they reach the script ORIENTI.

NO LOG CRT

\$(NC)

This stops the action of logging all displayed text. It is the opposite of \$(LC).

LOG DECOMP

\$(YZ)

Every word compared by DECOMP is logged. This is used for debugging the DECOMP function. It causes extensive logging and slows down the system. It should not be used until all other plans have failed.

STOP DECOMP LOG

\$(ZZ)

Do not log DECOMP words anymore. The opposite of \$(YZ).

Other functions are useful for debugging on-line, but do not directly cause special output on the log tape.

SCRIPT

\$(SC)

This causes the name of the current script and step to be displayed on the CRT.

LIST NAMES

\$(LN:X)

This function causes the names of all forms currently in memory to be displayed. Between each name the string X is inserted. Usually X is a carriage return.

PRINT FORM

\$(PF:N)

The form N is displayed with all of its segment gaps. Notation for segment gaps will vary with the implementation.

PRINT BINARY

\$(PB:N:D1:D2)

The binary string N is displayed as a series of ones and zeros. The starting point is bit D1. D2 bits are displayed. If D1 is omitted, display begins with the first bit. If D2 is omitted, it is assumed to be 255. Print binary allows you to examine strings bit by bit, an impossibility without fancy processing using other functions.

Section C9 Special Functions Not For Debugging

TELETYPE

#(TT)

This function indicates that the inquirer is on a teletype rather than on a video terminal. It suppresses the video formatting characters.

QUIT

#(QT)

This function allows you to exit from the GLURP system, but remain in time shared mode on-line. It should not be confused with the inquirer command @QUIT which causes the inquirer to log off but remain in the GLURP system ready for the next inquirer.

LOGIN

#(LI:X:Z)

User with code X is logged in if possible. This involves setting up the form NICK and establishing his access to the NICK file. If the login fails, the value of the function is Z.

LOGOUT

#(LO)

The current inquirer is terminated and the system is prepared to accept a new inquirer at that terminal.

APPENDIX A

SCRIPT TOPICS AND SOLUTIONS

SOME POSSIBLE SCRIPT TOPICS

1. Write a script segment that asks the inquirer to type in a pet's name and thanks him.
2. Write a script segment that asks the inquirer to type an odd digit (1, 3, 5, 7, or 9) into the machine and reprimands him for not doing it if he doesn't. The script should reinforce him if he does.
3. Write a script segment that asks the inquirer to type a word beginning with a vowel. Reinforce or teach him about vowels as appropriate.
4. Write a script that asks for any input, stores it, and then tells the inquirer if the input has an even or odd number of characters.
5. Write a script segment that makes secret code by substituting the alphabet reversed from the forwards version. For example, whenever there is an A, put a Z, for each B put a Y. Ask the inquirer for an input and report back the code to him.
6. Compute the sum of all the integers from 1 to N where the inquirer tells you N.

SOME SOLUTIONS

1. 100.00 CRT TYPE IN ANY NAME FOR A PET.
110.00 KEYBOARD
120.00 CRT+ THANKS.

2a. 100.00 CRT TYPE IN SOME ODD DIGIT.
110.00 KEYBOARD
120.00 DECOMP:(1 3 5 7 9):160.00
130.00 CRT NO, THAT IS NOT AN ODD DIGIT.
140.00 GOTO 200.00
160.00 CRT NICE WORK.
200.00 . . .

2b. change 120.00 in 2a. to read
DECOMP:1,3,5,7,9:160.00:160.00:160.00:160.00:160.00

2c. 100.00 CRT TYPE IN SOME ODD DIGIT.
110.00 KEYBOARD:140.00
115.00 DECOMP:(*1 3 5 7 9):160.00
120.00 CRT NO, THAT IS NOT AN ODD DIGIT
125.00 GOTO

4. 100.00 CRT TYPE IN ANY STATEMENT
110.00 KEYBOARD
120.00 #(DS:OLD:##(CK))
271.00 #(DS:NEW:(##(CN:OLD:2:*272.00)#(NEW)))#(NEW)
272.00 #(PS: #(EQ:##(CC:OLD)::EVEN:ODD) LENGTH)

5. ADVANCED SOLUTION

050.00 #(DS:ALPHABET:(A:B:C:D:E:F:G:H:I:J:K:L:M:N:O:P:Q:R:S:T:U:V:W:X:Y:Z))
060.00 #(DS:REVERSED:(Z:Y:X:W:V:U:T:S:R:Q:P:O:N:M:L:K:J:I:H:G:F:E:D:C:B:A))
100.00 CRT TELL ME SOMETHING TO ENCODE.
110.00 KEYBOARD
120.00 #(DS:ORIGINAL:##(CK))
130.00 #(SS:ORIGINAL:##(ALPHABET))
140.00 #(PS:##(ORIGINAL:##(REVERSED)))

6. ADVANCED SOLUTION

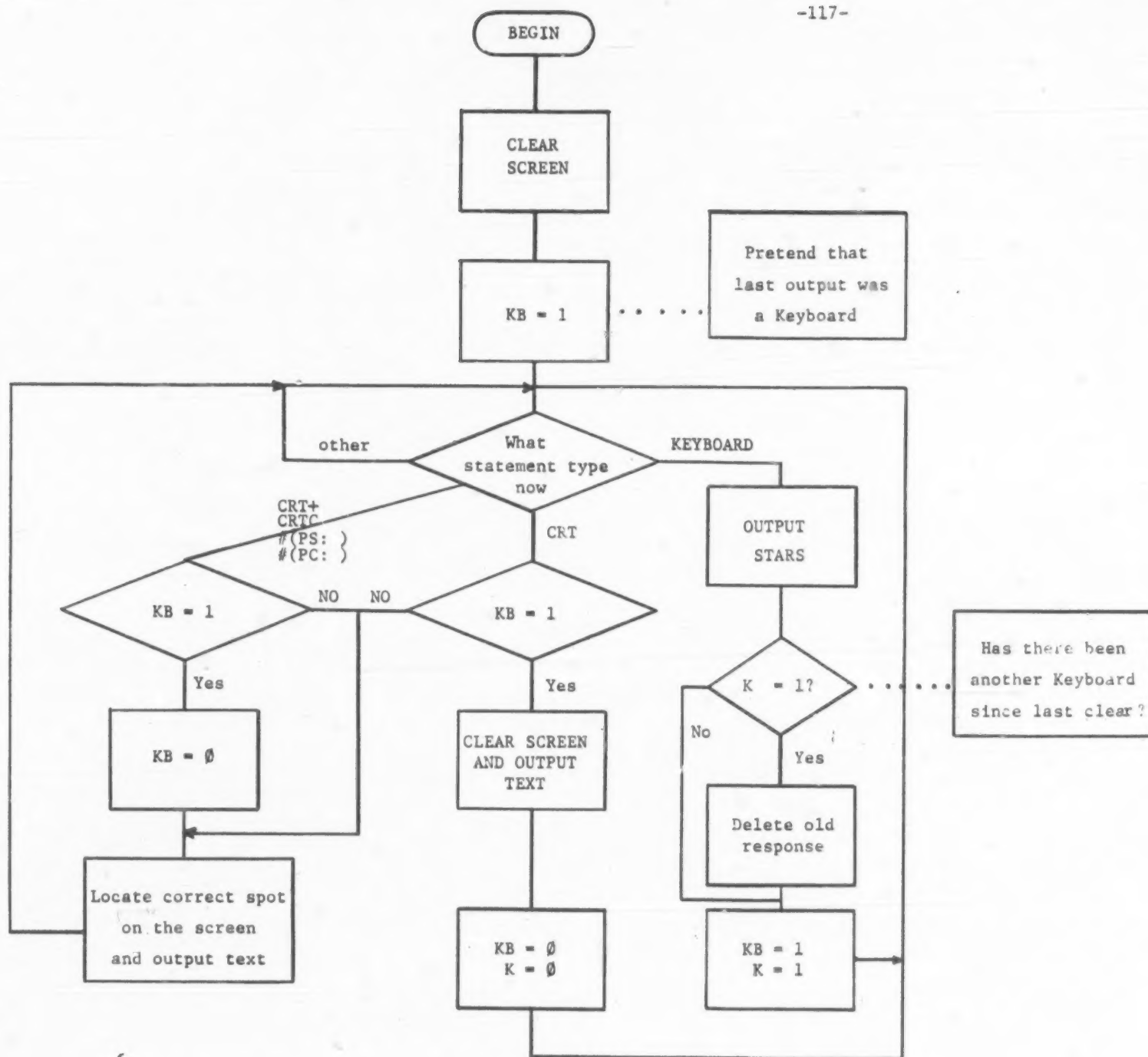
050.00 CRT TELL ME AN INTEGER
160.00 #(NUMBER:##(CK):500.00:200.00)
200.00 CRT THAT IS NO NUMBER. TRY AGAIN.
210.00 GOTO 050.00
500.00 #(PS:##(DV:##(ML:##(CK):##(AD:##(CK):1)):2))

6. [Note: Thus uses the formula for the sum, $N(N+1)/2$]

```
050.00 CRT TELL ME AN INTEGER
060.00 #(DS:SUM:0)#(DS:N:1)
070.00 #(AN:SUM:$(N))
100.00 #(AN:N:1)#(CP:$(N):$(CK):*070.00:*070.00:*200.00)
200.00 #(PS:THE SUM IS $(SUM))
```

APPENDIX B

FLOW CHART OF CRT DECISIONS IN GLURP



APPENDIX C

MULTI-PUNCHES FOR SANDERS FORMATTING CHARACTERS

		EBCDIC HEXADECIMAL	MULTIPLE PUNCHES	RCA SYMBOL
null		00	12,0,9,8,1	
F S	(front space)	08	12,9,8	
H T	(horizontal tab)	4A	12,8,2	¢
V T	(vertical tab)	4F	12,8,7	
H M	(home)	6A	12,11	^
C R	(carriage return)	5F	11,8,7	↵
B S	(back space)	18	11,9,8	
B T	(back tab)	19	11,9,8,1	
S B	(start blink)	BF	12,11,0,8,7	
C B	(clear blink)	70	12,11,0	
B R	(back return)	1D	11,9,8,5	
[		BB	12,11,0,8,3	
\		BC	12,11,0,8,4	
Move Cursor		B0	12,11,0,8,1	
Type		B1	12,11,0,1	
Format Type		B2	12,11,0,2	
Insert		B3	12,11,0,3	
Format Insert		B4	12,11,0,4	
Delete		B5	12,11,0,5	
Format Delete		B6	12,11,0,6	
Send Block		B7	12,11,0,7	
Send Page		B8	12,11,0,8	
C L	(clear)	B9	12,11,0,9	
Reset Cursor		BA	12,11,0,8,2	

APPENDIX D

ISVD RULE LISTS

QUEST

WHAT
WHY
WHERE
WHEN
HOW
WHO
WHICH

POS

YES	GROOVY
AOK	FINE
TRUE	GOOD
RIGHT	YEP
CORRECT	YUP
OK	SURE
OKAY	NATCH
OF COURSE	NATURALLY
CERTAINLY	ABSOLUTELY
YEAH	WHY NOT
YEA	SOCK IT TO ME
SI	AL RIGHT
POSITIVE\$	GUESS SO

UNSURE

DON'T KNOW
UNSURE
NOT SURE
NO IDEA
WHAT IS THERE
UNCERTAIN
MAYBE
WHAT CAN I DO
WHAT IS AVAILABLE
ANYTHING
WHATEVER
DO NOT KNOW
DON'T UNDERSTAND
DO NOT UNDERSTAND
DONT UNDERSTAND

NEG

NO
FALSE
INCORRECT
NOPE
NEVER
NEGATIVE
WRONG
NYET
NOT
DONT
WONT
CANT
CANNOT
NAW

WON'T
DON'T
ISN'T
AREN'T
DOESN'T
DIDN'T

APPENDIX E

THE WEIRD FUNCTIONS

Advanced users will want to examine Appendix E describing functions in detail. It is important to know which functions modify forms storage, which have values and which have none.

Notation in the appendix is standard:

N, N1, N2, N3, N4	name of a form
Z, Z1, Z2, Z3	any string without leading * or *stepno
STEPNO, STEPNO1, STEPNO2	ddd.dd where each d is a digit 0,1,2...9
FILE	1-8 character file name
RECORD	1-32 character record name
ATTRIBUTE	1-17 character attribute name
D, D1, D2	Numbers - any string of digits
X, X1, X2, X3	any string of characters

TRAC FUNCTIONS IN GLURP

Feb. 25, 1969

#(AD:D1:D2:Z)

Adds decimal numeric digits D1 to digits D2. A negative number is indicated by a leading minus sign as in '-3'. No decimal points may be included in the number. If the arithmetic capacity of the machine is exceeded, the value of the string is Z. Normally, the value of the string is D1 plus D2. Does not change forms storage.

#(AN:N:D:Z)

ADD AND STORE

This function is similar to AD, except that the first argument is a string name instead of a number. The contents of N are incremented by D. If the arithmetic capacity of the machine is exceeded, the value of the string is Z and forms storage remains unchanged. Normally, the value of the string is the contents of N plus D, and this value replaces the previous contents of N.

#(AORAN:X)

Permanent Form

A OR AN

The value of the function is 'AN X' if the first character of X is a, e, i, o, or u. Otherwise the value is 'A X'.

Any previous definition of A*** is deleted from forms storage. The form is as follows: (@1, @2, @3 indicate segment gaps.)

```
#(DS:A***:@1)
#(DS:A***:$(CC:A***))
#(SS:A***:A:E:I:O:U)
#(EQ:$(A***:::))$(DD:A***)::AN@1:A @1)
```

#(BA:N1:N2)

BINARY AND

This function performs a binary 'and' function of the binary string N2 and the binary contents of string N1. The results are left in N1 and the function has no value. For every position that is a 1 both in N1 and the contents of N2, a 1 is placed in that position of N1.

#(BC:N)

BINARY COMPLEMENT

This function reverse the sense of each position in binary string contained in N. Every 1 is changed to 0, every 0 is changed to 1. The result is stored in N. The function has no replacement value. In the ISVD version of BC, the number of bits is always a multiple of 8.

#(BE:N1:N2)

BINARY EXCLUSIVE OR

Performs a binary 'exclusive or' function with the contents of N1 and binary string N2. Every position that is a 1 in one string and a 0 in the other is set to 1 in N1. The results are stored in N1 and the function has no replacement value.

#(BO:N1:N2)

BINARY OR

Performs binary 'or' function with the contents of N1 and the binary string N2. Whenever a position has a 1 in either string, or both strings, a 1 is put in that position in string N1. The function has no replacement value.

#(CB:N)

COUNT BINARY BITS

The value is the number of 1's in binary string named N. Forms storage is unchanged.

#(CC:N:Z)

CALL CHARACTER

The value is the character under the form pointer. Every string has a counter that shows what part of a string is to be considered the current beginning. In this way the programmer can use pieces of a string, letting the machine keep track of where he left off. This is one of the functions that uses the string in this fashion. If, say, we had string N with contents

abcdefghijkl

the first use of the CC function would find the form pointer pointing at 'a' and the form pointer moved up to position 'b'. The next use of the CC instruction would produce the value 'b', unless some other form-pointer using function was used in between, of course. The CC function skips over segment gaps (ss SS function). If, in the example, the forms pointer were pointing to 'l', the next use of CC would produce the value 'l', but the next use of the function would produce the value Z.

#(CE:N1:N2:Z:N3)

COMPARE FOR EQUALITIES

In GLURP strings take on several forms. One of these is the word list, N, consists of a series of words separated by blanks. Each word in the string is considered an item of the list as is every comma.

This function compares word list N1 against word list N2. If all the words on N1 are on N2, the value of the function is Z and N3 remains unchanged in forms storage. If all the items on N1 are not on N2, then those items which are on both N1 and N2 are put in list N3. Any previous definition of N3 is deleted. The function normally has no replacement value.

#(CI:N1:N2:Z:N3)

COMPARE FOR INEQUALITIES

Similar to CE, except that those items which are on list N1 but not on list N2 are put on list N3. The function normally has no replacement value.

#(CK)

CALL KEYBOARD

The value of the function is the last student response to a KEYBOARD statement. Forms storage remains unchanged.

#(CL:N:X1:X2:.....)

CALL STRING

The value of this function is created by bringing in string N and filling in the segment gaps of value 1 with X1, segment gaps 2 with X2, etc. (See SS function.) Forms storage remains unchanged. In practice, this function is omitted and a 'call' is accomplished by using a string name in the function position instead of a function name

#(N:X1:X2:.....)

The colons for segment gaps must be present even if null arguments are presented.

#(CM:X) CHANGE META
 Changes the meta character to the first character of the string X. Upon starting the meta character is apostrophe. The meta character is used only when in 'programmer mode' or 'non-script' mode. It signals the machine to stop reading and analyze the input. In student mode the 'send block' button performs this function. The function has no replacement value.

#(CN:N:D:Z) CALL N CHARACTERS
 This is similar to CC, except that its value is the next D characters of string N. If D characters do not remain on the string, the value is Z and the form is unchanged. The forms pointer normally is moved to the letter after the D specified characters of the string.

#(CO:N1:N2:N3:N4) CONCATENATE
 This function creates a new string N1 which consists of the three string N2, N3, N4 put one next to the other. If one or more arguments are omitted, only those strings specified are used to create the new string, in the order specified. Any previous definition of N1 is deleted. The function has no replacement value. Strings N2, N3, and N4 are left unchanged.

#(COUNT:N) Permanent Form COUNT CHARACTERS
 The value is the number of characters in the string N.
 Previous definitions of A*** and B*** are deleted from forms storage. The form pointer to N is restored to the beginning, although the count is from the pointer to the end of the form. The form is as follows:

```

#(LOOP:A***:100:(#(***: #(CN:@1:10:(#(LOOP:B***:10:(
#(CC:@1:(#(CR:@1)#(COUNT***))))))))))
    
```

#(COUNT*)**

```

#(AD: #(A***):0: #(B***))#(DD:A***:B***)
    
```

#(CP:D1:D2:Z1:Z2:Z3) ARITHMETIC COMPARE
 Compares digits D1 with D2. If D1 is less than D2, the value of the function is Z1; if D1 is the same as D2, the value of the function is Z2; if D1 is greater than D2, the value is Z3. Forms storage is left unchanged. A null value is treated as zero.

#(CR:N) CALL RESTORE
 Restores the forms pointer back to the beginning of a string after it has been moved up by CC, CS, PT, etc. The function has no replacement value.

#(CS:N:Z) CALL SEGMENT
 The value of this function is the string from the current position of the form pointer to the next segment gap of string N. The value is the rest of the string if no segment gaps remain. If the string is empty, the value is Z. The form pointer is moved to the first character following the segment gap.

#(DC:FILE:ATTRIBUTE:N:Z) DECODE BINARY

The binary string named N is decoded according to the same table that would be used for ATTRIBUTE in FILE. Entries corresponding to the different bits are separated by commas. This string is the value of the function. Only 256 characters are allowed each time DC is used. If there are no more bits to decode, or the decode table is not found, the value is Z. Issuing the same function again causes decoding to continue where it left off.

#(DD:N1:N2:...) DELETE DEFINITION

Deletes strings N1, N2, etc. The function has no replacement value.

#(DE:FILE:ATTRIBUTE:N:Z) DECODE ITEM

The function is replaced by the decoded value of the contents of N. The decoding takes place as if the contents of N were the value of the attribute. If the decoding cannot take place, the value of the function is Z. Forms storage is unchanged.

#(DF:FILE:RECORD:ATTRIBUTE:N:Z) DELETE FROM DISC WITH FETCH

Deletes the data item specified by FILE, RECORD, ATTRIBUTE from the disc, at the same time inserting it in string N. If N is not specified, the value of the function is the deleted data; otherwise the function has no replacement value. If the data cannot be found, the value of the function is Z.

#(DI:N:X1:X2:...) DELETE ITEMS FROM LIST

In the list named N, items X1, X2, etc. are deleted. The function has no replacement value.

#(DN:N:D:Z) DIVIDE AND STORE

The contents of N are divided by D. If the arithmetic capacity of the machine is exceeded, the value of the string is Z and forms storage is unchanged. Normally, the value of the string is the contents of N divided by D rounded down to the next lower whole number, and this value replaces the previous contents of N.

#(DS:N:X) DEFINE STRING

Defines in forms storage a string named N with contents X. Function has no replacement value.

#(DT:FILE:RECORD:ATTRIBUTE::Z) DELETE FROM DISC

Deletes specified data item from disc. If record cannot be found or is not to be deleted because of its protection key, the value of the function is Z. Otherwise, the function has no replacement value. Forms storage remains unchanged.

#(DV:D1:D2:Z) DIVIDE

The value of the function is the quotient of D1 and D2, to the next lowest integer. If the arithmetic capacity of the machine is exceeded, the value of the string is Z. Forms storage is unchanged.

#(EC:X) **CHANGE ERASE CHARACTER**
 This function is also used only in programmer mode. Normally, the quote mark is used to erase its preceding character during input from the teletype. This can be changed with EC to the character specified in X. The function has no replacement value and the forms storage is unchanged.

#(ENDINS:X1:X2:X3) **Permanent Form** **END IN S**
 If the last character of the string X1 is the letter S, the value is X2, otherwise the value is X3. Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(DS:A***@1)
#(***: #(CS:A***))
#(EQ: #(CN:A***:-1):S:@2:@3) #(DD:A***)
```

#(EQ:X1:X2:Z1:Z2) **CHARACTER EQUALITY TEST**
 Compares string X1 with string X2. If they are equal, the value of the function is Z1, otherwise the value is Z2. Forms storage is unchanged.

#(FO:FILE:RECORD:ATTRIBUTE:N:Z) **FETCH ONLY**
 Gets the specified data item from the disc. If N is specified, the data is put in string N and the function has no replacement value. If N is not specified, the function value is the data item; if the data item cannot be found, the value of the function is Z. The item is not decoded.

#(FR:FILE:RECORD:ATTRIBUTE:N:Z) **FETCH RECORD WITH DECODING**
 Gets the specified data item from the disc. If N is specified, the data is put in string N and the function has no replacement value. If N is not specified, the function value is the data item; if the data item cannot be found, the value of the function is Z. Items is decoded according to the appropriate table. If no decoding is specified, the literal value is returned.

#(GO:STEPNO) **GOTO**
 Transfers script control to statement STEPNO. STEPNO must be in the format DDD.DD where all the D's are digits. This is different from the Z format, in that there is no leading *. The function has no replacement value and forms storage is untouched. Control is transferred immediately as soon as this function is recognized; any remaining functions in the current statement are ignored.

#(ID:FILE:RECORD:ATTRIBUTE:X:Z) **INSERT ON DISC**
 Inserts data item X on the disc with the specified labels. If the label is already represented on the disc, the item is not inserted and the value of the function is Z; otherwise, the function has no replacement value. Forms storage is unchanged.

#(IN:N:X:Z) **INITIAL**
 Starting from the form pointer, the form name is searched for the first location where X matches the string. The value of the function is the string from the pointer up to the character just before the matching string section. If a match is not found, the value is Z. The form pointer is moved to the character following the matching substring, or is not moved if there is no match.

#(KS:D) KEYBOARD SEGMENT
 The value of this function is the Dth segment of the last user response as interpreted by the last used DECOMP statement. The keyboard buffer and forms storage are unchanged.

#(LG:X) LOG MESSAGE
 The string X is written to the log tape. There is no replacement value and forms storage is unchanged.

#(LI:X:Z) LOGIN
 The session is initialized for inquirer X. If X is an unacceptable identification, the value is Z. After correct use, forms storage contains NICK, the inquirer's nickname and PASSWORD, the inquirer's password. Normally there is no replacement value.

#(LK:N) LOAD KEYBOARD
 The contents of the string N is placed in the keyboard buffer, and can then be used for DECOMP statements. Forms storage is unchanged. The function has no replacement value.

#(LN:X) LIST NAMES
 The value is a list of all the names of forms in forms storage separated by the string X. Forms storage is unchanged.

#(LO) LOGOUT
 The session is terminated for the inquirer issuing the function. All forms are deleted. There is no replacement value. No other inquirer is effected.

#(LOOP:N:D:X) Permanent Form LOOP THROUGH
 The string X is executed D times. The counter is kept in N in forms storage. If X causes a transfer of control, the value of N will be the number of times the loop was completed.
 The form is as follows:
 #(DS:@1:0)
 #(LOOP*: #(AD:@2:1): (@3))**
 LOOP***
 #(CP: #(AN:@1:1): @2: (@3#(LOOP*: @1: @2: (@3))))**

#(ML:D1:D2:Z) MULTIPLY
 Multiplies decimal digits D1 and D2. No decimal points are allowed. Normally, the value is the product of D1 and D2. If the arithmetic capacity of the machine is exceeded, the value is Z. Forms storage is unchanged.

#(MN:N:D:Z) MULTIPLY AND STORE
 This function is similar to ML, except that the first argument is a string name instead of a number. The contents of N are multiplied by D. If the arithmetic capacity of the machine is exceeded, the value of the string is Z and forms storage remains unchanged. Normally, the value of the string is the contents of N times D, and this value replaces the previous contents of N.

#(NB:N:X) NEW BOTTOM LIST
 Puts item X on the bottom of the list named N. The function has no replacement value.

#(NT:N:X) NEW TOP LIST
 Puts item X on the top of the list named N. If the form pointer has been advanced by the use of CC, CS, CN or PT, only the remaining portion of the list will be used with X at the top. The function has no replacement value.

#(NUMBER:X:STEPNO1:STEPNO2) Permanent Form NUMBER
 If X is all digits, control transfers to STEPNO1, otherwise control transfers to STEPNO2. Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(DS:A***:@1)
#(SS:A***:0:1:2:3:4:5:6:7:8:9)
#(EQ:#{A***:.....})#(DD:A***):(
    #(GO:@2)):
    #(GO:@3))
```

#(PB:N:D1:D2) Permanent Form PRINT BINARY
 The string N is displayed as a bit string on the screen, starting at bit D1, for D2 bits. Forms storage is unchanged and there is no replacement value. If D1 is omitted, it is assumed to be 1. If D2 is omitted, it is assumed to be 255. If D2 exceeds 255, it is assumed to be 255. Any previous definition of A*** is deleted from forms storage. The form is as follows:

```
#(DS:A***:#{EQ:@2::0:#{SU:@2:1}})
#(PS:#{LOOP***:A***:
    #(CP:@3:256:#{EQ:@3::256:@3}):256:@3):
    #{TB:@1:#{A***:1:0}}))
#(DD:A***)
```

#(PC:X) PRINT CONTINUOUS
 The string X is displayed at the next available space. There is no replacement value, and forms storage is unchanged.

#(PF:N) PRINT FORM
 Prints the contents of the string named N, including indications of any segment gaps. The function has no replacement value and forms storage remains unchanged.

#(PL:N:X) PRINT LIST
 Prints each word in word list N separated by the character(s) X. The function has no replacement value and forms storage is unchanged. If X is a carriage return, the list N is printed in a column.

#(PS:X) PRINT STRING
 Prints string X on the next available line. The function has no replacement value and forms storage remains unchanged.

#(PT:N:Z) POP TOP LIST
 The value of the function is the next item from the top of word list N. If no words remain, the value is Z. The form pointer is advanced to the next item in the list. Multiple spaces are treated as one space.

#(RC) READ CHARACTER
 Reads the next character from the teletype (CRT) and assumes that value.
 This function is used only in programmer mode.

#(RD:FILE:RECORD:ATTRIBUTE:X:Z) REPLACE ON DISC
 Replaces the data item specified by data X. If the existing item cannot be found, no replacement takes place and the value of the function is Z. Otherwise the function has no replacement value. Forms storage is unchanged.

#(RS) READ STRING
 Reads characters from the console until it finds a meta character, and assumes this read in string as its value. This function is used in programmer mode only.

#(RT:STEPNO) RESET STEP
 Sets the script branch control at STEPNO, but does not transfer control until the rest of the current statement is analyzed. STEPNO must be in the format DDD.DD where all the D's are digits. There is no leading * as there is in the Z format.

#(SC) SCRIPT CURRENT
 The script name and step number are displayed on the screen. Forms storage is unchanged, and there is no replacement value.

#(SD:X) SELECTIVE DELETE
 Deletes from form storage all strings whose names begin with the character string X. The function has no replacement value.

#(SLIDE:X:MSS) Permanent Form DISPLAY SLIDE
 The slide named X is displayed on the slide screen. If the slide is unavailable, a description appears on the CRT. If the wrong tray is mounted, a message to change it is sent. The slide is displayed for M minutes and SS seconds. If the last argument is blank, the slide is displayed for ten seconds. If there is a concurrent message, it is displayed at the same time as the slide. Forms storage is unchanged, and there is no replacement value.

Any previous definitions of A***, B***, T***, ID*** are deleted. FILE is expected to contain the name of the current file. When the command is finished, TRAY1*** has the name of the tray mounted on projector 1, and TRAY2*** has the name of the tray mounted on projector 2. PROJ*** has the number of the currently used projector. PR1*** has a count of recent usage of projector 1 and PR2*** has a count of recent usage of projector 2. The form is as follows:

```

#(DS:ID***:@1)#(DS:T***:@2)
#(SC1*)#(DD:T***:ID***:A***:B***)
```

SC1*

```

#(EQ: #(FO: #(FILE): #(ID***): VLBL): 1: (#(SC2*))): (
  #(TB: IOERR: 11: (
    #(PS: #(FO: #(FILE): #(ID***): DSCRPTN))))))
```


SC2*

```
#(EQ: #(TRAY1***): #(FO: #(FILE): #(ID***): TRY: A***): #(A***): (
  #(AN: PR1***: 1) #(DS: PROJ***: 1) #(SC3*)): (
  #(EQ: #(TRAY2***): #(A***): (
    #(AN: PR2***: 1) #(DS: PROJ***: 2) #(SC3*)): (#(MOUNT))))
```

SC3*

```
#(DD: A***) #(PS: >\ SO#(PROJ***)/
  #(EQ: #(T***): : 010: (#T***))) /
  #(FO: #(FILE): #(ID***): PSTNS9: : ) /// * [
  #(FO: #(FILE): #(ID***): CNCRRNTM) <
  > \ SO#(PROJ***) /// * < )
  #(PS: PRESS THE RUN BUTTON NOW.)
```

MOUNT

```
#(PS: MOUNT SLIDE TRAY #(A***) ON PROJECTOR
  #(CP: #(PR1***): #(PR2***): (#(DS: B***: 1)): (
    #(DS: B***: 1)): (#(DS: B***: 2))) #(B***))
  #(DS: PR1***: 0) #(DS: PR2***: 0) #(DS: PR#(B***): ***: 1)
  #(DS: TRAY#(B***): ***: #(A***)) #(SC3*))
```

#(SN: N: D: Z)

SUBTRACT AND STORE

Subtracts D from the contents of string N and replaces the previous contents of N with this sum. The value of the function is the new contents of N. If the arithmetic capacity of the machine is exceeded, the value of the function is Z and forms storage is unchanged.

#(SR: FILE: RECORD: ATTRIBUTE: X: Z)

STORE RECORD ON DISC

Stores data X on disc with specified label. If label already exists on disc, the old one is deleted and the new item put on instead. The function has no replacement value and forms storage is unchanged.

#(SS: N: X1: X2: ...)

SEGMENT STRING

The string named N is scanned from left to right with respect to string X1. If a substring is found matching X1, that substring is removed from string N and marked with a segment marker of value 1. The marker is called a 'segment gap'. The rest of the string N is scanned with respect to X1 to create any additional segment gaps. The parts of the string not taken by segment gaps are now scanned with respect to X2, and the matching substrings, if any, are marked as segment gaps of value 2, etc. The untouched portions of the resulting strings are called 'segments'.

#(SU: D1: D2: Z)

SUBTRACT

The value of the function is the result of subtracting the number D2 from D1. If the result exceeds the arithmetic capacity of the machine, the value of the function is Z. Forms storage remains unchanged.

#(TAKEN: D: STEPNO1: STEPNO2) Permanent Form

TAKEN THE SCRIPT

If the inquirer has taken the script numbered D, control transfers to STEPNO1, otherwise control transfers to STEPNO2.

If STEPNO2 is omitted, the next step is taken. Any previous definition of A*** is deleted from forms storage.

The form is as follows:

```

#(FO:#(NICK):TAKEN:ETAKEN:A***:(#(DD:A***)))
#(BO:A***:STAKEN)
#(TB:A***:@1:(#(DD:A***)#(GO:@2)):(
    #(DD:A***)
    #(EQ:@3::(#(GO:@3))))))

```

#(TB:N:D:Z1:Z2) TEST BINARY BIT
 Tests the Dth binary bit in the binary string named N. If that bit is a 1, the value is Z1; otherwise it is Z2.

#(TF:N:D) TURN OFF BINARY BIT
 Forces bit D of binary string named N to be 0. The function has no replacement value.

#(TO:N:D) TURN ON BINARY BIT
 Forces bit D of binary string named N to be 1. The function has no replacement value.

#(TT) TELETYPE
 This function tells the program that the user is on the teletype, so that the formatting characters are appropriate.

#(WD:N:Z) Permanent Form WORD
 The value of the function is the next word list N. If no word remains, the value is Z. The form pointer is advanced to the next item on the list. Word delimiters include . ? , ; : ! and space.

Any previous definition of A*** is deleted from forms storage.
 The form is as follows:

```

#(DS:A***:#(CS:@1:(#(GO:@2))))
#(SS:A***:, : . ; : ? : ( : ) : ! )
#(DS:A***:##(CS:A***))
##(IN:@1:#(A***))
#(CC:@1)#(A***)#(DD:A***)

```

#(XQ:SCRIPTNAME:STEPNO) EXECUTE
 Sends control to script SCRIPTNAME, statement STEPNO. If STEPNO is not designated, control goes to the first statement of that script. Control is transferred immediately, without further analysis of the current statement. Forms storage is left unchanged. If the step or script does not exist, control is passed through the STOP script to the script on top of the LINK list.

APPENDIX F

THE TRAC ALGORITHM

TRAC, A Procedure-Describing Language for the Reactive Typewriter

Calvin N. Mooers

Rockford Research Institute Inc., Cambridge, Massachusetts

A description of the TRAC (Text Reckoning And Compiling) language and processing algorithm is given. The TRAC language was developed as the basis of a software package for the reactive typewriter. In the TRAC language, one can write procedures for accepting, naming and storing any character string from the typewriter; for modifying any string in any way; for treating any string at any time as an executable procedure, or as a name, or as text; and for printing out any string. The TRAC language is based upon an extension and generalization to character strings of the programming concept of the "macro." Through the ability of TRAC to accept and store definitions of procedures, the capabilities of the language can be indefinitely extended. TRAC can handle iterative and recursive procedures, and can deal with character strings, integers and Boolean vector variables.

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965; a supplementary paper, not on the announced program.

The present work was supported in part by the following grants and contracts: Advanced Research Projects Agency contract SD-295, Information Sciences Branch of the Air Force Office of Scientific Research contracts AF-AFOSR 376, 377 and 461-64, and Division of General Medicine, National Institutes of Health, U.S. Public Health Service grant GM 10416.

Introduction

The TRAC (Text Reckoning And Compiling) language system is a user language for control of the computer and storage parts of a reactive typewriter system. A reactive typewriter is understood to be one of a number of teletypewriters simultaneously connected online by wire to a memory and computer complex which permits real-time, multiple access (time-shared) operation. In the philosophy of the reactive typewriter, the man at the typewriter keyboard is the focal point of the system. The connected storage and computer devices are considered to be peripheral service units to the reactive typewriter.

The design goals for the TRAC language and its translating system included: (1) high capability in dealing with back-and-forth communication between a man at a keyboard and his work on the machine, so as to permit him to make insertions and interventions during the running of his work; (2) maximum versatility in the definition and performance of any well-defined procedure on text; (3) ability to define, store and subsequently use such procedures to extend the capabilities of the language; and finally (4) maximum clarity in the language itself so that it could be easily taught to others. A discussion of these design goals, and of the design decisions which went into

the language, may be found in a companion paper [1]. The TRAC language has now been programmed for several computers.¹ It has shown a high degree of stability during the past year of experimental use.

The present TRAC language is machine-independent and is closed with respect to operations performed upon sets of characters from a typewriter keyboard. The present language should be precisely designated as "TRAC 64."² Later versions of TRAC are expected to have the ability to deal with strings of machine-coded words and with subroutines, and will thus be self-implementing.

The TRAC language was developed after a study of a number of procedure-describing languages, but only after it was concluded that each of these had features which were believed to be unsuitable or unduly constraining for the purposes contemplated by TRAC. In particular, the languages IPL-V, LISP and COMIT were carefully examined. In brief, IPL-V appeared to be too closely oriented to computer programming. LISP had severe restrictions due to its "atomic" symbols, certain conceptual confusions and too great an orientation to mathematical logic. COMIT, while suitable in many respects, had the rigidity associated with a compiler. When work was begun on TRAC (1960) none appeared to have the capabilities desired, though they were a definite source of inspiration.

The prime stimulus to the present TRAC language came from two important unpublished papers by Eastwood and McIlroy [3] and McIlroy [4]. The first paper described a macro assembly system having run-time definition-making and decision-making capabilities. The second paper showed how this system could perform very general manipulations on symbol strings. TRAC is a refinement and extension of the macro approach of these papers. Therefore it can be said that the present TRAC system consists of a machine-independent language together with a generalized macro text processor which runs interpretively to provide versatile interaction capabilities at run time. A recent, independently developed system by Strachey [5] has a number of remarkable similarities to TRAC.

TRAC Syntax

A TRAC string may contain a substring enclosed by a matching pair of parentheses, such as $(\dots)$ where the dots indicate a string. The matching parentheses indicate the scope of some particular action. There are three cases, epitomized by $\ast(\dots)$, $\ast\ast(\dots)$ and $(\dots)$.³ The first two formats indicate the presence of a TRAC "primitive function." The format $\ast(\dots)$ denotes an "active function," while the format $\ast\ast(\dots)$ denotes a "neutral function." This distinction is clarified below. The string interior to either kind of function is generally divided into substrings by commas as in $\ast(, ,)$ where these substrings constitute the arguments of the function. Parentheses in the format $(\dots)$ have roughly the same

¹ The computers are the Digital Equipment PDP-1, PDP-5, the General Electric Datatnet-30 and the Scientific Data Systems SDS 930.

² The present paper is a revision and extension of [2].

³ With the model 33 Teletype, the "up arrow" character is preferred instead of the sharp sign.

role as paired quotation marks, and, in particular, whatever string is inside the paired parentheses is protected from functional evaluation.

TRAC strings are dealt with by the TRAC processor according to a scanning algorithm which works from left to right and performs the evaluation of nested expressions from inside outward. In the expression

$$\frac{1}{4} \left(\frac{1}{3} \left(\frac{1}{1} \left(\frac{1}{2} \left(\frac{1}{1} \right) \right) \right) \right)$$

the functions are evaluated in the order indicated. As each function is evaluated, it is replaced in the TRAC string by the string (possibly null) which is its value. The evaluation of an active function is followed directly by the evaluation of any function in its value string not protected by matched parentheses. The value string of a neutral function is not further evaluated.

Examples of Functions

An example of the “define string” primitive function is the expression $\ast(ds, AA, CAT)$. This causes recording of the string *CAT* in the memory and places the name *AA* of the string in a table of contents. The string can be called out of memory by the “call” function $\ast(cl, AA)$. The result of the call function is to place the string value of the call, namely *CAT*, in the former location of $\ast(cl, AA)$, with an expansion or a closing up of the surrounding strings. The call function is in the class of functions having a “value string.” The define string function is an example of a function having a “null value”; i.e., no string is left behind in its place after its evaluation.

Evaluation of the "read string" function $\ast(rs)$ causes the processor to accept input from the typewriter. Its value is the string as received from the typewriter up to a terminating "meta character" which is usually taken to be the apostrophe. The meta character can be changed. The "print string" function $\ast(ps, X)$ causes printing out of the argument string, here represented by the symbol X . It has null value. The nested expression $\ast(ps, \ast(cl, A.I))$ will cause *CAT* to be printed out.

In the beginning, and at the completion of every processing cycle, the TRAC "idling procedure" $\ast(ps, \ast(i))$ is automatically loaded into the TRAC processor. It is therefore seen that all strings and programs are effectively loaded into the interior of the idling procedure, and furthermore, all TRAC computations are made on functions nested within the argument string of some other function.

TRAC Algorithm

The TRAC algorithm governs the precise manner in which TRAC expressions are scanned and evaluated by the TRAC processor. At the beginning, the unevaluated strings are in the "active string" and the "scanning pointer" points to the leftmost character in this string. As characters have been treated by the scanning algorithm, they may be added to the right-hand end of a "neutral string," which is so called because its characters have been fully treated by the algorithm and are thus neutral, like alphabetic characters. The algorithm follows.

1. The character under the scanning pointer is examined. If there is no character left (active string empty), go to rule 14.
2. If the character just examined (by rule 1) is a begin parenthesis, the character is deleted and the pointer is moved ahead to the character following the first matching end parenthesis. The end parenthesis is deleted and all nondeleted characters passed over (including nested parentheses) are put into the neutral string without change. Go to rule 1.
3. If the character just examined is either a carriage return, a line feed or a tabulate, the character is deleted. Go to rule 15.
4. If the character just examined is a comma, it is deleted. The location following the right-hand character at the end of the neutral string, called the "current location," is marked by a pointer to indicate the end of an argument substring and the beginning of a new argument substring. Go to rule 15.
5. If the character is a sharp sign, the next character is inspected. If this is a begin parenthesis, the beginning of an active function is indicated. The sharp sign and begin parenthesis are deleted and the current location in the neutral string is marked to indicate the beginning of an active function and the beginning of an argument substring. The scanning pointer is moved to the character following the deleted parenthesis. Go to rule 1.
6. If the character is a sharp sign and the next character is also a sharp sign, the second-following character is inspected. If this is a begin parenthesis, the beginning of a neutral function is indicated. Two sharp signs and the begin parenthesis are deleted and the current location in the neutral string is marked to indicate the beginning of a neutral function and the beginning of an argument substring. The scanning pointer is moved to the character following the deleted parenthesis. Go to rule 1.
7. If the character is a sharp sign, but neither rule 5 or 6 applies, the character is added to the neutral string. Go to rule 15.
8. If the character is an end parenthesis, the character is deleted. The current location in the neutral string is marked by a pointer to indicate the end of an argument substring and the end of a function. The pointer to the beginning of the current function is now retrieved. The complete set of argument substrings for the function have now been defined. The action indicated for the function is performed. Go to rule 10.
9. If the character meets the test of none of the rules 2 through 8, transfer the character to the right-hand end of the neutral string and go to rule 15.
10. If the function has null value, go to rule 13.
11. If the function was an active function, the value string is inserted to the left of (preceding) the first unscanned character in the active string. The scanning pointer is reset so as to point to the location preceding the first character of the new value string. Go to rule 13.
12. If the function was a neutral function, the value string is inserted in the neutral string with its first charac-

ter being put in the location pointed to by the current begin-of-function pointer. Delete the argument and function pointers back to the begin-of-function pointer. The scanning pointer is not reset. Go to rule 15.

13. Delete the argument and function pointers back to the begin-of-function pointer for the function just evaluated, resetting the current location to this point. Go to rule 15.

14. Delete the neutral string, initialize its pointers, reload a new copy of the idling procedure into the active string, reset the scanning pointer to the beginning of the idling procedure, and go to rule 1.

15. Move the scanning pointer ahead to the next character. Go to rule 1.

The TRAC processor will accept any string of symbols. Nonexistent functions are given a null value. Omitted arguments are given a null value, while extra arguments are ignored. Omitted right parentheses will cause the processor to terminate its action and reinitialize itself at an unexpected point, while extra right parentheses are ignored and deleted at the end of a procedure. When the processor becomes too full, perhaps due to an infinite iteration or recursion, a diagnostic is typed out to indicate that fact and the processor is re-initialized by going to rule 14. The break key stops any action and causes re-initialization.

The TRAC Functions

Input-Output. All functions are shown in their active representation, which is the form most often used. As shown, the argument strings are presumed not to contain functions or other active matter.

**(rs)* "read string" (one argument). (Note that the mnemonic for the function name is counted as the first argument.) The value is the string as read from the teletypewriter keyboard up to the point of occurrence of the meta character, which is deleted.

**(rc)* "read character" (one argument). The value is the next character, which may be any character (including the meta character) received from the teletypewriter.

**(cm,X)* "change meta" (two arguments). This null-valued function changes the meta character to the first character of the string symbolized by *X*. Upon starting, the TRAC processor is loaded with a standard meta character, usually the apostrophe.

**(ps,X)* "print string" (two arguments). This null-valued function prints out on the teletypewriter the string represented by *X*.

Define and Call Functions

**(ds,N,X)* "define string" (three arguments). This is a null-valued function. The string symbolized by *X* is placed in storage and is given the name symbolized by *N*. The name is placed in a name list or table of contents to the "forms" in storage. A "form" is a named string in storage. If a form is already in storage with name *N*, this form is erased. The name *N* may be a null string.

**(ss,N,X1,X2,...)* "segment string" (three or more arguments). This is a null-valued function. The form

named N is taken from storage and is scanned from left to right with respect to string $X1$. If a substring is found matching $X1$, the location of the match is marked. The matching substring is excluded from further action, thus creating a "segment gap." The rest of the form is scanned with respect to $X1$ to create any additional segment gaps. These segment gaps are all given the ordinal value one. The parts of the form not taken by segment gaps are now scanned with respect to string $X2$, with the creation of segment gaps of ordinal value two. This action is repeated with all of the remaining argument strings. At the end, the marked form, along with its pointers and ordinal identifiers for the segment gaps, is put back into storage with the name N . The untouched portions of the string in the form are called "segments." It is seen that the segment string function creates a "macro" in which the arguments $X1$, $X2$, etc., indicate the dummy variables. The segment string function subsequently can be applied with other arguments to the same form, with the result of new segment gaps being created with ordinal value one, two, etc., and being inserted among those already there. A null string for one of the arguments X causes no action for this argument.

*($cl, N, X1, X2, \dots$) "call" (two or more arguments). The value is generated by bringing the form named N from storage and filling the segment gaps of ordinal value one with string $X1$, the gaps of ordinal value two with string $X2$, and so on for all the segment gaps in the form.

The following specialized calls read out a part of a form. They treat the segment gaps as if the gaps were filled with the null string. These calls (cs , cc , cn , and in) preserve the neutral-active function distinction only for the strings coming from the form named N . Since the alternative value of these functions, symbolized by Z , may be a call to a procedure, the alternative value is always treated as if the function were active.

All the call functions (cl , cs , cc , cn , and in) read the text of a form beginning at the location indicated by a "form pointer" which is part of the apparatus of the form. Initially the form pointer points at the first character of the form. The call function does not change the form pointer.

*(cs, N, Z) "call segment" (three arguments). The value of this function is the string from the current location of the form pointer to the next segment gap of the form named N . If the form is empty, the value is Z . The form pointer is moved to the first character following the segment gap.

*(cc, N, Z) "call character" (three arguments). The value is the character under the form pointer. If the form is empty, the value is Z . The form pointer is moved one character ahead (segment gaps are skipped).

*(cn, N, D, Z) "call n characters" (four arguments). This function reads from the form named N from the point indicated by the form pointer and continuing for a number of characters specified by the decimal integer number at the tail end of the string symbolized by D . Segment gaps are skipped. If the decimal number is

positive, this function reads the string to the right of the pointer; if negative, to the left. The strings so read are preserved in their character sequence. If no characters are available to be read, the value is Z . The form pointer is moved (right or left) to the next unread character.

*(in, N, X, Z) "initial" (four arguments). Starting from the form pointer, the form named N is searched for the first location where the string X produces a match. The value is the string from the pointer up to the character just before the matching string. If a match is not found, the value is Z . The form pointer is moved to the character following the matching substring, or is not moved if there is no match.

*(cr, N) "call restore" (two arguments). This null-valued function restores the form pointer of the form named N to the initial character.

*($dd, N1, N2, \dots$) "delete definition" (two or more arguments). This null-valued function deletes the forms named $N1$, $N2$, etc., from memory and removes their names from the list of names.

*(da) "delete all" (one argument). This null-valued function deletes all the forms in memory, and removes their names.

Arithmetic Functions. TRAC does integer arithmetic, taking decimal arguments. The decimal numeric digits are looked for at the tail ends of the argument strings. The prefix string of the first argument string is preserved and is appended to the answer, while the prefix string of the second argument is ignored. Negative quantities are indicated by the minus sign "-", and initial zeros are ignored. Whenever the integer values become so large as to overflow the capacity of the arithmetic processor, the overflow value Z of the function is taken. The overflow value is always treated as if it were produced by an active function. The arithmetic functions are: *($ad, D1, D2, Z$) "add", *($su, D1, D2, Z$) "subtract", *($ml, D1, D2, Z$) "multiply" and *($dv, D1, D2, Z$) "divide". They all take four arguments. In these functions, $D2$ is subtracted from $D1$, and $D1$ is divided by $D2$, with the answer being the largest integer contained in the dividend.

Boolean Functions. Boolean TRAC functions operate on strings of bits (of value 0 or 1), i.e., on Boolean vectors. The bit strings are represented by octal digits, with each digit representing three bits. Thus the bit strings have lengths in multiples of three. The octal digits are looked for at the tail end of the 01 and 02 strings, and any non-octal prefix matter is deleted. The functions are: *($bu, 01, 02$) "Boolean union," *($bi, 01, 02$) "Boolean intersection," *($bc, 01$) "Boolean complement," *($bs, D1, 01$) "Boolean shift" and *($br, D1, 01$) "Boolean rotate." The bit strings are right justified. In the Boolean union the shorter string is filled out with leading zeros, while in the Boolean intersection, the longer string is truncated at the left. In the complement, shift and rotate, the length of the bit string remains the same. Shift is to the left by the number of places specified by the decimal $D1$ (with leading nondecimal matter being deleted) when

$D1$ is positive, and to the right when $D1$ is negative. The new positions created by the shift are filled with zeros. Rotate is also to the left or right, with positive or negative values of $D1$. The digits displaced from one end of the vector are added to the place created at the other end.

Decision Functions

$\#(eq, X1, X2, X3, X4)$ "equals" (five arguments). This is a test for string equality. If $X1$ is equal to $X2$, the value is $X3$; otherwise it is $X4$.

$\#(gr, D1, D2, X1, X2)$ "greater than" (five arguments). This is a test of numerical magnitude. If the integer decimal number at the tail of string $D1$ is algebraically greater than the number at the tail of $D2$ the value is $X1$; otherwise it is $X2$.

External Storage Management Functions

$\#(sb, N, N1, N2, \dots)$ "store block" (three or more arguments). This null-valued function assembles the group of forms named $N1, N2$, etc., and stores them as a block in an external storage area. The form names, segment gaps, etc., are all preserved. When the forms have been put into the external storage, they are erased from form storage. A new form is created with name N and with a string which is the address of the block in external storage.

$\#(fb, N)$ "fetch block" (two arguments). This null-valued function is the converse of the store block function. The name N is the name of the block to be fetched. The function restores to form storage all the forms in the block, complete with names, segment gaps, pointers, etc. It does not erase the block in external storage, nor the form named N .

$\#(eb, N)$ "erase block" (two arguments). This null-valued function erases the form named N and also the group of forms in the block in external storage.

These functions permit forms to be moved to and from the main memory and also protect the stored forms from accidental erasure. They also permit one to build a "storage tree." By this technique, a group of forms can be stored under a group name, a set of group names can be stored under a section name, and so on.

Diagnostic Functions

$\#(ln, X)$ "list names" (two arguments). The value of this function is the list of names in the name list, i.e., the names of all the forms in form storage. Each name in the value string is preceded by string X . If X is the character pair "carriage return, line feed" protected by double parentheses, the names will be listed in a column.

$\#(pf, N)$ "print form" (two arguments). This causes the typing out of the form named N with a complete indication of the location and ordinal values of the segment gaps.

$\#(tn)$ "trace on" (one argument). This null-valued function initiates the trace mode in which, as the computation progresses, the neutral strings for each function are typed out. Typing the backspace key causes evaluation of the function, and presentation of the neutral strings of the next. Typing anything other than backspace causes

initialization. Carriage return may be used instead of backspace.

$\#(tf)$ "trace off" (one argument). This is a null-valued function which terminates the trace mode without initialization. Both trace on and trace off functions may be placed anywhere in a procedure.

Examples of TRAC Procedures

1. The distinction between active and neutral functions is usually puzzling. In essence, the value from an active function is rescanned, while in the neutral function it is not. The following example shows the action of the protective parenthesis, the neutral and the active forms of the function. Consider that both $\#(ds, AA, CAT)'$ and the simple program $\#(ds, BB, (\#(cl, AA)))'$ have been presented to the processor. Then,

$\#(ps, (\#(cl, BB)))'$, $\#(ps, \#(cl, BB))'$, $\#(ps, \#(cl, BB))'$
prints out, respectively:

$\#(cl, BB)$, $\#(cl, AA)$, CAT

2. When the processor is quite full, it is often desirable to delete all forms but one of a particular name. The procedure $\#(ds, N, \#(cl, N) \#(da))$ will accomplish this. Here $\#(cl, N)$ reads the form N into the processor, and it is held in the neutral string while all the forms in memory are erased. The form is then redefined with its original name. In this example, segment gaps are lost.

3. This and the following example illustrate the extension of TRAC capabilities through defining and storing of suitable procedures. The calculation of the factorial of a number can be done by simple recursion:

$\#(ds, Factorial, (\#(eq, 1, X, 1,$
 $(\#(ml, X, \#(cl, Factorial, \#(ad, X, -1))))$
 $))) \#(ss, Factorial, X)'$

Then the call $\#(cl, Factorial, 5)'$ produces the result 120.

4. Many users will prefer to have TRAC supply its own sharp signs and parentheses when calling a procedure. The following will do this:

$\#(ds, English, (\#(ps,$
 $\#(cl, \#(rs)) ($
 $)) \#(cl, English)))'$

To start this action, we use $\#(cl, English)'$, and then if one types in $Factorial, 5'$ the response is 120 followed by carriage return, line feed. The action is terminated by $\#(dd, English)'$.

Acknowledgments. Special thanks are due to my collaborator L. Peter Deutsch for his assistance in the development and implementation of TRAC.

REFERENCES

1. MOOERS, C. N. TRAC, a text handling language. Proc. ACM 20th Nat. Conf. Cleveland, Aug. 1965, pp. 229-246.
2. TRAC—a procedure defining and executing system. Mem. V-157, Rockford Research, Cambridge, June 1964.
3. EASTWOOD, D. E., AND McILROY, M. D. Macro compiler modification of SAP. Mem., Comput. Lab., Bell Telephone Labs., Murray Hill, N.J., Sept. 3, 1959. (Unpublished)
4. McILROY, M. D. Using SAP macro instructions to manipulate symbolic expressions. Mem., Comput. Lab., Bell Telephone Labs., Murray Hill, N.J., 1960. (Unpublished)
5. STRACHEY, C. A general purpose macrogenerator. *Comput. J.* 8, 3 (1966).

DEVELOPED BY CALVIN N. MOGERS

• IS THE SCANNING POINTER

A. 15. 200. 100. 50. 25. 10. 5. 2. 1. 0.5. 0.25. 0.125. 0.0625. 0.03125. 0.015625. 0.0078125. 0.00390625. 0.001953125. 0.0009765625. 0.00048828125. 0.000244140625. 0.0001220703125. 0.00006103515625. 0.000030517578125. 0.0000152587890625. 0.00000762939453125. 0.000003814697265625. 0.0000019073486328125. 0.00000095367431640625. 0.000000476837158203125. 0.0000002384185791015625. 0.00000011920928955078125. 0.000000059604644775390625. 0.0000000298023223876953125. 0.00000001490116119384765625. 0.000000007450580596923828125. 0.0000000037252902984619140625. 0.00000000186264514923095703125. 0.000000000931322574615478515625. 0.0000000004656612873077392578125. 0.00000000023283064365386962890625. 0.000000000116415321826934814453125. 0.0000000000582076609134674072265625. 0.00000000002910383045673370361328125. 0.000000000014551915228366851806640625. 0.0000000000072759576141834259033203125. 0.00000000000363797880709171295166015625. 0.000000000001818989403545856475830078125. 0.0000000000009094947017729282379150390625. 0.00000000000045474735088646411895751953125. 0.000000000000227373675443232059478759765625. 0.0000000000001136868377216160297393798828125. 0.00000000000005684341886080801486968994140625. 0.000000000000028421709430404007434844970703125. 0.0000000000000142108547152020037174224853515625. 0.00000000000000710542735760100185871124267578125. 0.000000000000003552713678800500929355621337890625. 0.000000000000001776356839400250464677810668953125. 0.0000000000000008881784197001252323389053344765625. 0.00000000000000044408920985006261616945266723828125. 0.000000000000000222044604925031308084726333619140625. 0.0000000000000001110223024625156540423631668095703125. 0.00000000000000005551115123125782702118158340478515625. 0.000000000000000027755575615628913510590791702392578125. 0.0000000000000000138777878078144567552953958511962890625. 0.00000000000000000693889390390722837764769792559814453125. 0.000000000000000003469446951953614188823848962799072265625. 0.0000000000000000017347234759768070944119244813995361328125. 0.00000000000000000086736173798840354720596124069976806640625. 0.000000000000000000433680868994201773602980620349884033203125. 0.0000000000000000002168404344971008868014903101749420166015625. 0.00000000000000000010842021724855044340074515508747100830078125. 0.000000000000000000054210108624275221700372577543735504150390625. 0.0000000000000000000271050543121376108501862887718677520751953125. 0.00000000000000000001355252715606880542509314438593387603759765625. 0.000000000000000000006776263578034402712546572192966938018798828125. 0.0000000000000000000033881317890172013562732860964834690093994140625. 0.00000000000000000000169406589450860067813664304824173450469970703125. 0.0000000000000000000008470329472543003390683215241220672523499853515625. 0.00000000000000000000042351647362715016953416076206103362617499267578125. 0.000000000000000000000211758236813575084767080381030516813087496337890625. 0.000000000000000000000105879118406787542383540190515258406543748168953125. 0.0000000000000000000000529395592033937711917700952576292032718740844765625. 0.00000000000000000000002646977960169688559588504762881460163593704223828125. 0.000000000000000000000013234889800848442797942523814407300817968521119140625. 0.0000000000000000000000066174449004242213989712619072036504089842605595703125. 0.00000000000000000000000330872245021211069948563095360182520449213027978515625. 0.000000000000000000000001654361225106055349742815476800912602246065139892578125. 0.0000000000000000000000008271806125530276748714077384004563011230325699462890625. 0.00000000000000000000000041359030627651383743570386920022815056151628497314453125. 0.000000000000000000000000206795153138256918717851934600114075280758142486572265625. 0.0000000000000000000000001033975765691284593589259673000570376403790712432861328125. 0.00000000000000000000000005169878828456422967946298365002851882018953562164306640625. 0.000000000000000000000000025849394142282114839731491825014259410094767810821533203125. 0.0000000000000000000000000129246970711410574198657459125071297050473839054107666015625. 0.000000

A IS THE BALL STAYING IN THE
B

IS AN ARBITRARY FLO

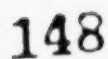
C.R. IS CARRIAGE RETURN

LF IS LINE FEED

TAB IS TABULATE

ONE CAN THINK OF THE ACTIVE STRING AS BEING A PART
OF THE NEUTRAL STRING, SO THAT CHARGES ARE TRANSMITTED
ACROSS A GAP IN RETURN.

THE IDLING PROCEDURE 179 (1991, p. 145)



BIBLIOGRAPHY

- Feingold, S.L.; and Frye, C.H. (1966) "User's Guide to PLANIT: Programming LANguage for Interactive Teaching." Tech Memo-3055/000/01. Santa Monica, California: System Development Corporation.
- Mooers, Calvin. (March 1966) "TRAC, A Procedure-Describing Language for the Reactive Typewriter." Communications of the ACM. 9,3.
- Roman, Richard A. (December 1969) "Developing and Implementing Materials for Computer Assisted Instruction." Project Report No. 26. Cambridge, Massachusetts: Information System for Vocational Decisions, Harvard Graduate School of Education.
- Sanders Associates. (1966) User's System Description Manual for Model 720 Communicator System. Nashua, New Hampshire.
- Taylor, Ann. (1970) "GLURP -- GENERALIZED LANGUAGE for UNDERSTANDING and RESPONDING to PEOPLE." Project Report No. 29. Cambridge, Massachusetts: Information System for Vocational Decisions, Harvard Graduate School of Education.
- Weizenbaum, Joseph. (1966) "ELIZA -- A Computer Program for the Study of Natural Language Communication Between Man and Machine." Communications of the ACM. 9,1.
- Yee, Patricia; Roman, Richard A.; Little, Priscilla A.; and Archibald, David K. (1970) "The Construction and Implementation of a Data Base." Project Report No. 30. Cambridge, Massachusetts: Information System for Vocational Decisions, Harvard Graduate School of Education.

INDEX OF FUNCTIONS

(AD)	Add	.58, 130
(AN)	Add and Store	.130
(AORAN)	A or An	.52, 77, 130
(BA)	Binary And	93, 130
(BC)	Binary Complement	.92, 130
(BE)	Binary Exclusive Or	.94, 130
(BO)	Binary Or	.94, 130
(CB)	Count Binary Bits	.92, 131
(CC)	Call Character	61, 131
(CE)	Compare for Equalities	69, 131
(CI)	Compare for Inequalities	.69, 131
(CK)	Call Keyboard	.50, 131
(CL)	Call String	.131
(CM)	Change Meta	.132
(CN)	Call N Characters	.64, 132
(CO)	Concatenate	.71, 132
(COUNT)	Count Characters	.78 132
(CP)	Arithmetic Compare	49, 132
(CR)	Call Restore	65, 132
(CS)	Call Segment	65, 132
(DC)	Decode Binary	103, 133
(DD)	Delete Definition	.71, 133
(DE)	Decode Item	105, 133
(DF)	Delete from Disc with Fetch	100, 133
(DI)	Delete Items from List	67, 133
(DN)	Divide and Store	133
(DS)	Define String	.33, 45, 133
(DT)	Delete from Disc	99, 133
(DV)	Divide	58, 133
(EC)	Change Erase Character	134
(ENDINS)	End in S	77, 134
(EQ)	Character Equality Test	.43, 134
(FO)	Fetch Only	.103, 134

(FR)	Fetch Record with Decoding	.103, 134
(GO)	GOTO	73, 134
(ID)	Insert on Disc	99, 134
(IN)	Initial	.64, 134
(KS)	Keyboard Segment	50, 135
(LC)	Log CRT	.106
(LG)	Log Message	106, 135
(LI)	LOGIN	108, 135
(LK)	Load Keyboard	.76, 135
(LN)	List Names	.107, 135
(LO)	LOGOUT	.108, 135
(LOOP)	Loop Through	78, 135
(ML)	Multiply	58, 135
(MN)	Multiply and Store	135
(NB)	New Bottom List	.66, 136
(NC)	No Log CRT	106
(NT)	New Top List	66, 136
(NUMBER)	Number	52, 136
(PB)	Print Binary	.107, 136
(PC)	Print Continuous	72, 136
(PF)	Print Form	.107, 136
(PL)	Print List	72, 136
(PS)	Print String	40, 72, 136
(PT)	Pop Top List	67, 136
(QT)	Quit	108
(RC)	Read Character	137
(RD)	Replace on Disc	.99, 137
(RS)	Read String	137
(RT)	Reset Step	74, 137
(SC)	Script Current	.107, 137
(SD)	Selective Delete	71, 137
(SLIDE)	Display Slide	39, 137
(SN)	Subtract and Store	138
(SR)	Store Record on Disc	98, 138

(SS)	Segment String	138
(SU)	Subtract	58, 138
(TAKEN)	Taken the Script	.52, 77, 138
(TB)	Test Binary Bit	.91, 139
(TF)	Turn Off Binary Bit	.91, 139
(TO)	Turn On Binary Bit	90, 139
(TT)	Teletype	.108, 139
(WD)	Word	139
(XQ)	Execute	.75, 139
(YZ)	Log DECOMP	107
(ZZ)	Stop DECOMP Log	.107